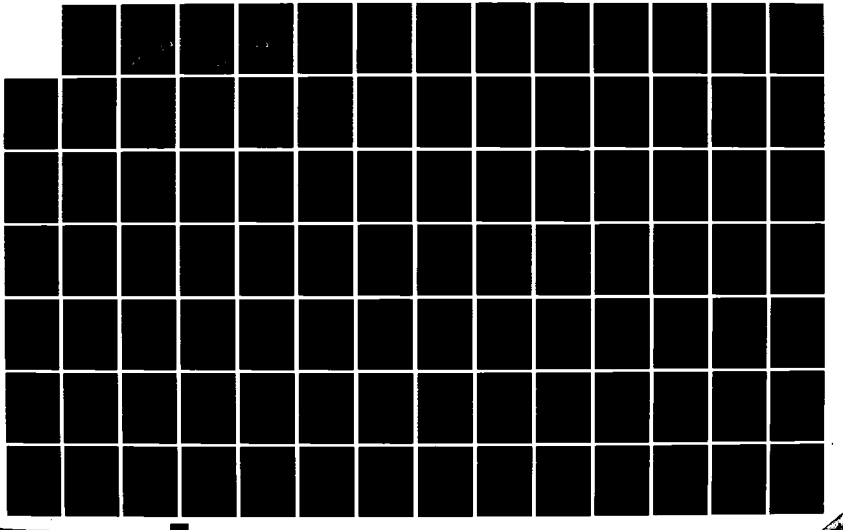
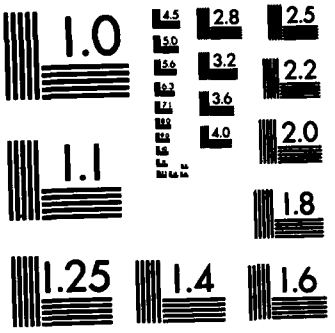
AD-A123 307 LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 1/2
STORE AND FORWARD MES..(U) GENERAL DYNAMICS FORT WORTH
TX DATA SYSTEMS DIV 09 NOV 82 DAAK80-81-C-0108
UNCLASSIFIED F/G 17/2 NL






MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

AD A123307

CASE STUDY I

# FINAL REPORT DEVELOPED FOR LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MESSAGE SWITCH USING THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME IV OF IV







GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101

CASE STUDY I

# FINAL REPORT DEVELOPED FOR LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MESSAGE SWITCH USING THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME IV OF IV

GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101

50272-101

| REPORT DOCUMENTATION PAGE | 1. REPORT NO. | 2. | 3. Recipient's Accession No. AD-A123 307 |
|---|---|---|---|

**4. Title and Subtitle**

Ada Capability Study: Design of the Message Switching System AN/TYC-39 Using the Ada Programming Language

**5. Report Date**

November 1982

**6.**

**7. Author(s)**

General Dynamics

**8. Performing Organization Rept. No.**

**9. Performing Organization Name and Address**

General Dynamics
Data Systems Division
Central Center
P. O. Box 748
Fort Worth, TX 76101

**10. Project/Task/Work Unit No.**

**11. Contract(C) or Grant(G) No.**

(C) DAAK80-81-C-0108

(G)

**12. Sponsoring Organization Name and Address**

USA CECOM
Center for Tactical Computer Systems (CENTACS)
ATTN: DRSEL-TCS-ADA-1
Fort Monmouth, NJ 07703

**13. Type of Report & Period Covered**

Final

**14.**

**15. Supplementary Notes**

**16. Abstract (Limit: 200 words)**

An Ada oriented framework for the design and documentation of the U. S. Army TYC-39 store and forward message switch (military software) system is presented. This document package contains a Requirements, Design, Ada Integrated Methodology, and Final Report section. A methodology to use Ada in specifying requirements, design, and the implementation of a system was developed. This methodology was used to redesign the TYC-39 message switch system. A selected software module was programmed after the redesign.



**17. Document Analysis a. Descriptors**

Ada Programming Language
Software Design with Ada
Designing with Ada

**b. Identifiers/Open-Ended Terms**

Message Switch
Military Software
Program Design Language

**c. COSATI Field/Group**

**18. Availability Statement:**

Distribution limited to the United States. Available from National Technical Information Service, Springfield, VA 22161.

| 19. Security Class (This Report) | 21. No. of Pages |
|---|---|
| UNCLASSIFIED | 521 |
| **20. Security Class (This Page)** | **22. Price** |
| UNCLASSIFIED | |

TABLE OF CONTENTS


| | | PAGE |
|---|---|---|
| 1. | Introduction | 1 |
| 2. | System Design | |
| | 2.1 System Hierarchy | 3 |
| | 2.2 Run Switch | 5 |
| | 2.2.1 Overview | 5 |
| | 2.2.2 Operation | 6 |
| | 2.2.3 Functions | 9 |
| | 2.3 Internal Message Structure | 20 |
| | 2.4 Functional Decomposition and Interconnectivity | 22 |
| | 2.4.1 Message Input | 22 |
| | 2.4.2 Message Queueing | 42 |
| | 2.4.3 Message Routing | 47 |
| | 2.4.4 Output Message | 56 |
| | 2.4.5 Manage Intransit and Manage Overflow | 65 |
| | 2.5 Data Structure Diagrams | 68 |
| 3. | Detail Design | |
| | 3.1 Ada Unit Specification | 77 |
| | 3.1.1 Message Input | 77 |
| | 3.1.2 Message Queueing | 81 |
| | 3.1.3 Message Routing | 86 |
| | 3.1.4 Output Message | 88 |
| | 3.1.5 Manage Intransit and Manage Overvlow | 91 |
| | 3.1.6 Support Routines | 94 |
| | 3.2 Package Specification Dependencies | 115 |
| | 3.3 Traceability | 117 |
| | 3.4 Data Structures | 127 |
| | 3.5 Rationale for Hardware/Software Partitioning | 136 |
| 4. | Detail Hardware Design | |
| | 4.1 General Configuration | 141 |
| | 4.2 Serial Data Bus | 143 |
| | 4.3 Line Termination Unit | 143 |
| | 4.4 Design Features | 145 |
| 5. | Implementation of Selected Function (Output Message) | |
| | 5.1 Hardware Implementation | 146 |
| | 5.2 LTU Processor PCB | 146 |
| | 5.3 LTU Channel PCB | 146 |
| | 5.4 Software Implementation | 150 |

## LIST OF FIGURES


| Figure No. | Description | Page |
|---|---|---|
| 2.1-1 | System Hierarchy Chart | 4 |
| 2.2-1/4 | Run Switch Hierarchy Charts | 16 |
| 2.3-1 | Message Schema | 21 |
| 2.4-1 | Receive Valid Message | 23 |
| 2.4-2 | Receive Valid Message Interconnectivity | 24 |
| 2.4-3 | Build Valid Asynchronous Segment | 25 |
| 2.4-4/6 | Build Valid Asynchronous Segment Interconnectivity | 26 |
| 2.4-7 | Build Synchronous Segment | 29 |
| 2.4-8 | Build Synchronous Segment Interconnectivity | 30 |
| 2.4-9 | Process MCB | 31 |
| 2.4-10 | Process MCB Interconnectivity | 32 |
| 2.4-11 | Data Chart for Process MCB | 33 |
| 2.4-12 | Validate Message | 34 |
| 2.4-13 | Validate JANAP Message | 35 |
| 2.4-14 | Validate ACP Message | 36 |
| 2.4-15 | Validate HPJ Message | 37 |
| 2.4-16 | Validate HPA Message | 38 |
| 2.4-17 | Validate Message Interconnectivity | 39 |
| 2.4-18 | Abbreviation Chart for Validate Message | 40 |
| 2.4-19 | Data Chart for Validate Message | 41 |
| 2.4-20 | Manage Message Queues | 43 |
| 2.4-21 | Manage Message Queues Interconnectivity | 44 |
| 2.4-22 | Manage Translated Message Queues | 45 |
| 2.4-23 | Manage Translated Message Queues Interconnectivity | 46 |
| 2.4-24 | Process Message | 48 |
| 2.4-25 | Process Message Interconnectivity | 49 |
| 2.4-26 | Route Message | 50 |
| 2.4-27 | Route Message Interconnectivity | 51 |
| 2.4-28 | Segregate Routing Line Interconnectivity | 52 |
| 2.4-29 | Complete Version Header Interconnectivity | 53 |
| 2.4-30 | Translate Message | 54 |
| 2.4-31 | Translate Message Interconnectivity | 55 |
| 2.4-32 | Output Message | 57 |
| 2.4-33 | Output Message Interconnectivity | 58 |
| 2.4-34 | Send ModeI Interconnectivity | 59 |
| 2.4-35 | Send Mode V and Send Mode II & IV Interconnectivity | 60 |
| 2.4-36 | Header Valid | 61 |
| 2.4-37 | Header Valid Interconnectivity | 62 |
| 2.4-38 | Decouple Log | 63 |
| 2.4-39 | Decouple Log Interconnectivity | 64 |
| 2.4-40 | Message Storage Allocation/Deallocation | 66 |
| 2.4-41 | Message Storage Allocation/Deallocation Interconnectivity | 67 |

LIST OF FIGURES
(cont.)


| Figure No. | Description | Page |
|---|---|---|
| 2.5-1 | Journal Object | 69 |
| 2.5-2 | Reference Storage Object | 70 |
| 2.5-3 | Phsyical Port Object | 71 |
| 2.5-4 | Message Object | 72 |
| 2.5-5 | Logical Port Table and Queue Objects | 73 |
| 2.5-6 | Route Table Object | 74 |
| 2.5-7 | Example of Message Input Utilizing Object Oriented Structure | 75 |
| 2.5-8 | Example of Process Message Utilizing Object Oriented Structure | 76 |
| 3.2-1 | Package Specification Dependencies | 116 |
| 3.5-1 | Logical Result of Hardware/Software Partitioning | 138 |
| 3.5-2 | Output Message Main Processor Software Diagram | 139 |
| 3.5-3 | Output Message Remote Processor Software Diagram | 140 |
| 4.1-1 | Messge Switch Hardware Block Diagram | 142 |
| 4.3-1 | LTU Block Diagram | 144 |
| 5.2-1 | LTU Processor PCB Block Diagram | 147 |
| 5.3-1 | LTU Channel PCB Block Diagram | 149 |

## 1. Introduction

This document is a description of the final Army TYC-39 Message Switch design using the "Ada Integrated Methodology" developed by General Dynamics Data Systems Division. Although the rationale for the design decisions is presented here, the issues that were debated in order to arrive at this design are discussed in the project "Final Report". The basis for this design is contained in another document, "Ada Equivalent System Requirements Specification", which explains the functional operation of a message switch and compatible interface equipment. All three of the documents mentioned are separately submitted as a part of this project.

The design description is organized into four major sections, which are: 1) System Design, 2) Detail Design, 3) Detail Hardware Design and 4) Implementation. This project was accomplished without any preconceived ideas about hardware/software partitioning. Therefore, the entire system design was completed before the partitioning was done. Then, because of the limited scope of the project, the "message output" section was chosen for detailed design and implementation.

The "System Design" section contains charts and diagrams from which the structure of the message switch was derived. The charts begin with the system hierarchy, which shows the major packages required for message processing. Run switch is presented next because of the order in which the system must be started and initialized. Once up and running with a loaded database, the message processing can begin. Most of the design effort was concentrated in this area because of the critical real time nature of this process. The internal message storage structure is presented in the message schema and the detailed logic of message processing is shown in the functional decomposition diagrams. The data structure diagrams are included because much of the resulting design is based on the ideas derived from these early design sessions (see final report).

The "Detail Design" section provides more specific details of system operation. The Ada Unit Specifications were developed for the structure of the entire switch before considering hardware/software partitioning, as were the data structures, traceability, and package specification dependency chart. Then the analysis that was given to system partitioning is presented.

The "Detail Hardware Design" provides some information required to implement the system as partitioned so that the messages can be processed within the constraints of the non-functional requirements. Since "message output" was selected for implementation, only the input/output section of the message switch hardware detail is shown. The input processing at the line termination unit is so tightly coupled

to message output that both processes had to be done together. Because a multiprocessing environment was chosen, the detailed hardware section describes the added complication of interprocessor operations.

Finally, the "Implementation" section describes the lowest level block diagram of the Line Termination Unit and an explanation of the intended method to dynamically allocate the tasks required for message output. Due to its size, the Ada source code is contained in another volume entitled, "Source Code Document".

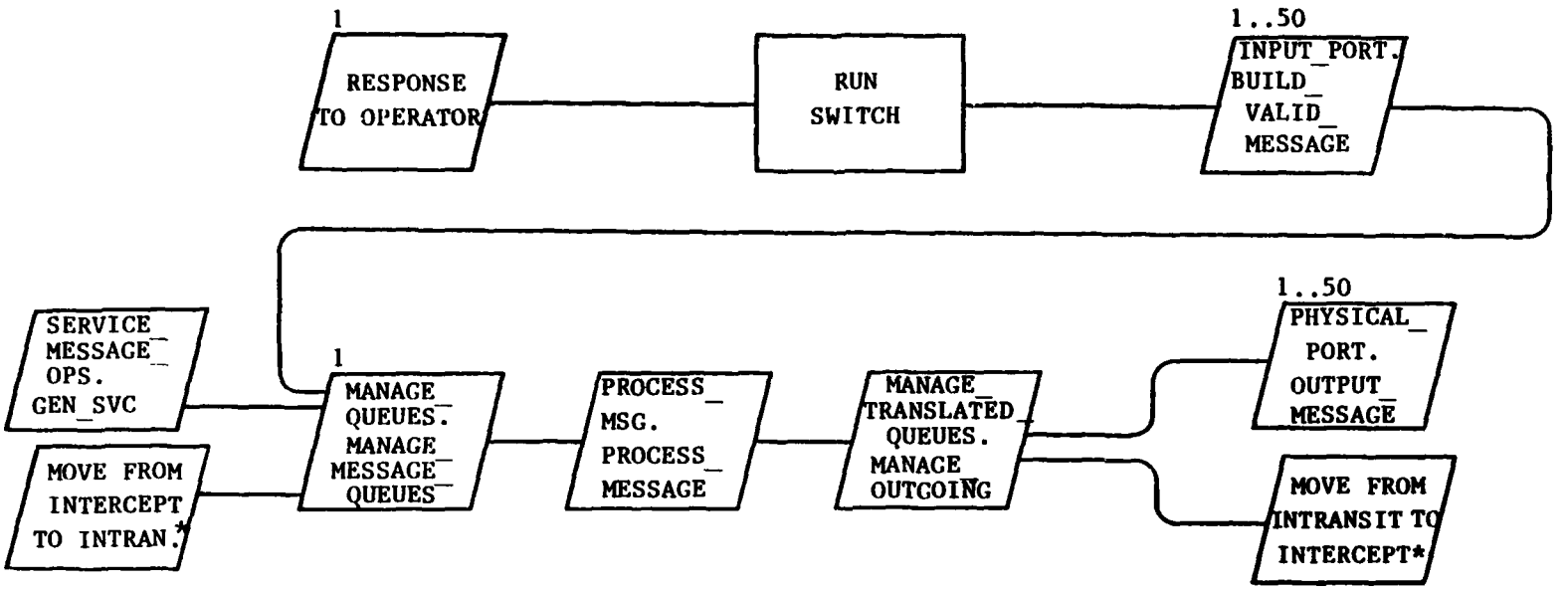
## 2. System Design

This section contains system charts and diagrams representing design decisions based on the "Ada Equivalent System Requirements Specification for the AN-TYC-39 Store and Forward Message Switch". The design proceeded according to a methodology tailored for this project and described in the document "Ada Integrated Methodology".

### 2.1 System Hierarchy

The chart shown in Figure 2.1-1 shows the top level operation of the message switch. Run Switch and Response to Operator are functions necessary for supervision of the message processing. They are described in the Run Switch Section, but not to the detail required by an implementation. The scope of the project was somewhat limited, thus this aspect being less time critical, a minimal effort was applied to this task.

The other parallelograms presented represent Ada tasks with the name before the period being the package name containing the task. The name after the period is the task name if the figure is a parallelogram or a procedure name if the figure is a rectangle. The internal memory structure required to store the large volumes of data handled by the message switch is shown by the Manage Intransit and Manage Overflow diagram contained in section 2.4.5. These tasks and the History Ops package are utilized as necessary out of the other operational tasks.




* MANAGE_TRANSLATED_QUEUES.
  MOVE_INTERCEPT

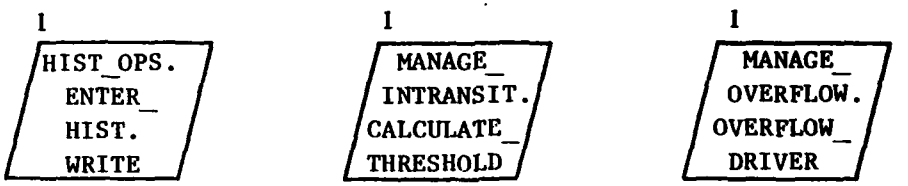



Figure 2.1-1 HIERARCHY CHART

## 2.2 Run Switch

### 2.2.1 Overview

In any large system design there are multitudes of details which must be considered and coordinated in order to produce the most efficient system. Since the primary purpose of this contract was to develop methods of using Ada and to investigate its applicability to a communication system, some areas of the system design were considered in less detail than others. These areas were chosen for two reasons: 1) they were standard problems which were not particularly related to Ada or to communications systems, and 2) to reduce the complexity of the job and allow indepth design and analysis of unique Ada applications. Although Run Switch, including the Operator Interface, was selected as an area less deserving of detailed examination, in the interest of thoroughness and integration with the rest of the system design some work was done to describe the basic structure and define th required operations. The following Description and Theory of Operation is an overview of Run Switch and is not intended to be a full treatment of the problem.

### 2.2.2 Operation

Run Switch is the program unit which initiates, monitors and terminates switch operation. In order to accomplish these functions, Run Switch contains various sub-program units in the relationships illustrated in sheets 1-4 of the Run Switch diagram and described below. The software design does not make use of Run Switch as an operating system or an executive. Ada provides the operating system and an executive is unnecessary. Once initiated, the switch software will be either self driven or externally driven rather than depending on an executive. Run Switch will simply monitor the operation.

A. Initiation - Initiation consists of the capability to begin switch operation under one of three possible start-up conditions: 1) Cold Start, 2) Restart, after an ordered (Normal) shut-down, or 3) Recovery, after an emergency or unscheduled shut-down. Although all three operations achieve the same results (an operational switch with valid tables and status and an open log), the methods by which these results are achieved are different because of different initial conditions.

1. Start Switch - This is the "cold-start" sequence and assumes a lack of reliance on previous operating information. This situation might be encountered after a move to a new location or after switch reconfiguration. Provisions are made for operator entry of table and status information as well as use of prestored information. Since previous messages and journal information are either not available or not used, these items are initially set to the null state, in preparation for actual switch operation.

2. Restart Switch - This sequence assumes resumption of switch operation after an orderly shut-down. In this case, the operating tables, status information, and the final balanced journal would have been saved on non-volatile media. Restart would then retrieve this information and request any changes from the operator. The final journal would be the basis for the new journal and saved messages would be retrieved from the long history file and checked. The retrieved messages would be written into Intransit Storage.

3. Recover Switch - This sequence is slightly different from Restart in that the previous shut-

down was either unscheduled or an emergency and the normal, ordered shut-down process was not followed. In this case, all of the required information may not be available in the desired format, and some preliminary processing may be required. For example, the journal will be available from non-volatile media but in an emergency shut-down time may not have been available to balance it. Therefore the balancing must be done to maintain accountability, before normal switch operation resumes. Tables and status information do not present the same problems because in normal operation this information is periodically saved, as well as updated when changes are made. Of course, the operator will be able to change the tables or status as conditions warrant, after they are reloaded.

B. Operation - During normal switch operation, Run Switch will be concerned with monitoring the operation of the switch and maintaining the operator interface. This also includes monitoring of current status and potential hardware and software error conditions, as well as performing background diagnostics and periodically balancing the log.

C. Termination - Termination of switch operation can take place in either of two forms: 1) Normal (ordered) Shut-Down or 2) Emergency (unscheduled) Shut-Down. The normal sequence is more desirable, however, it cannot be assured under all circumstances.

1. Stop Switch (Normal) - This sequence assumes that sufficient time is available to complete an orderly shut-down. Since this time is a minimum only, the DRY-UP Command is a sub-set of, or a preliminary to a Normal Stop. Depending on the time available, message reception and transmission could be terminated as soon as the current message on a channel is finished (Fast Dry-Up), or reception could be terminated with receipt of the current message and transmission on a channel terminated after all messages available for that channel have been transmitted. In any case, after these conditions are met, generation and saving on non-volatile media of final journal and other operating information would be accomplished. At this point the switch is no longer operational, although the operator interface portion of run switch will continue to run, allowing off line diagnostics, data base changes, etc. From this state, power could be cut off with no detrimental effects.

2. Stop Switch (Emergency) - This sequence causes immediate termination of message I/O, and if time is available, updates the [illegible]g. Since the assumption of this sequence is that the required time for a normal shut-down is not available, the sequence attempts to accomplish only essential items. It is anticipated that the sequence may be entered by operator command or upon automatic detection of a catastrophic fault (such as power failure).

### 2.2.3 Functions

Reference Figure 2.2-1

A. Load Program - controls the loading of the nonresident portions of the switch software from nonvolatile storage.

B. Get RI Table - determines if the RI Table should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as appropriate.

   1. RI_OPS.INIT_RI_TABLE - prepares the RI Table storage space for the entry of new information.

   2. RI_OPS.LOAD_RI_TABLE - retrieves the prestored or saved RI Table data from nonvolatile storage and loads that information into the RI Table storage space.

   3. Prompt Operator for Changes - initiates an interactive mode for operator verification of loaded RI Table information and entry or modification of information as required.

   4. RI_OPS.SAVE_RI_TABLE - duplicates the Current RI Table in nonvolatile storage.

C. Get Line Table - determines if the Line Table should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as required.

   1. LINE_TBL_OPS.INIT - prepares the Line Table storage space for the entry of new information.

   2. LINE_TBL_OPS.LOAD - retrieves the prestored or saved Line Table data from nonvolatile storage and loads that information into the Line Table storage space.

   3. Prompt Operator for Changes - initiates an interactive mode for operator verification of loaded Line Table information and entry or modification of information as required.

   4. LINE_TBL_OPS.SAVE - duplicates the current Line Table in nonvolatile storage.

Figure 2.2-1 (continued)

D. Get Status - determines if the Status information should be retrieved from nonvolatile storage or requested from the operator, and initiates the following actions as required.

   1. Initialize Status - prepares the Status storage space for the entry of new information.

   2. Load Status - retrieves the prestored or saved Status from nonvolatile storage and loads that information into the Status storage space.

   3. Prompt Operator for Changes - initiates an interactive mode for operator verification of loaded Status information and entry or modification of information as required.

   4. Save Status - duplicates the current Status in nonvolatile storage.

E. AUDIT.AUDIT_INTRANSIT - used during recovery to read the previous log and generate an audit which will become the basis for the new Running Audit. Also produces a list of messages to be recovered from Reference Storage and reintroduced.

F. HISTORY_OPS.REENTER_MESSAGES - accepts a list of messages to be recovered, attempts to recover those messages and reintroduce them into the system. If necessary, generates a list of messages which could not be recovered.

G. HISTORY_OPS.INIT_HISTORY - begins a new History with no initial entries. Used when there is no previous Journal.

H. Activate Channels - controls and coordinates the activation of the individual communication channels.

   1. Initializes Channels - prepares the channels for message transmission and reception by specifying the necessary operating controls and parameters for each individual channel.

   2. Generate Service Messages - as appropriate, Service Messages will be generated for transmission to the channels' distant ends to indicate that a link has been activated.

   3. Generate Control Characters - as appropriate, Control Characters will be generated for transmission to the Channel's distant ends to indicate that a link has been activated.

4. Save Status - the operational status and operating parameters of all channels will be saved in nonvolatile storage.

Figure 2.2-2

I. Monitor Hardware Errors - This section provides for software acceptance of monitoring outputs. It also will initiate alarms, corrective action, or operator notification as appropriate. Automatic hardware error checking and monitoring is assumed.

J. Monitor Software Errors - provide for the detection, monitoring, and correction of software errors as well as tabulating and correcting (when possible) exceptions raised by the software, perform independent error detection, and initiate the appropriate action for the error condition.

K. Monitor Status - changes in equipment status will be automatically monitored. This may cause automatic switchovers for backed-up equipment as well as alarms and notices to the operator. Operator entered status changes will be saved for future reference.

L. Perform Background Diagnostics - this task will be performed continuously during switch operation to test both operational capabilities and error detection capabilities.

1. Save - Periodically saves in nonvolatile storage the results of monitoring hardware, software and status and of Background Diagnostics.

2. Print - initiates the hard copy output of detected error conditions and the periodic printing of the status of the monitored conditions.

M. HISTORY_OPS.NEW_DAY - a standard package that closes the old day's history and saves the current audit, begins a new day's history, then audits the old day's history and compares the generated audit to the saved running audit. Also compiles the statistics for the old day, initiates printing the statistics and clears the file for the new day.

Figure 2.2-3

N. Interface to Operator - provides coordination between the various subfunctions of the operator interface. These subfunctions interact with the operator as well as initiating the requested operation(s). The interactive portions will be of the menu driven, selection and response type with automatic verification before activation.

1. Command Interface - coordinates the activities required for command acceptance before routing the command to the appropriate subfunction for execution.

   a. Verify Command - insures that an operator entered command is correct and valid for the context in which it is being saved.

   b. Interpret Command - converts the operator entered command from a series of key strokes to a selection code with appropriate parameters for routing to and use by the other subfunctions.

2. Control RI Table - depending on the selection code and parameters which initiate this subfunction, the appropriate entry point to the RI_OPS Package is selected.

   a. Read - provisions to allow the operator to examine or verify the entries in the RI Table.

   b. Modify - provisions to allow the changing, addition or deletion of any or all of the RI Table entries.

   c. Save - initiates the storage of the RI Table in nonvolatile storage for future reference.

3. Control Line Table - depending on the selection code and parameters which initiate this subfunction, the appropriate entry point to the LINE-OPS Package is selected.

   a. Read - provisions to allow the operator to examine or verify the entries in the Line Table.

   b. Modify - provisions to allow the changing, addition or deletion of any or all of the Line Table entries.

   c. Save - initiates the storage of the Line Table in nonvolatile storage for future reference.

4. Trace Message - provides the capability to search the journal and retrieve all log entries pertaining to a particular message for presentation to the operator.

   a. Search - initiates a sequential journal search to locate the next log entry for a particular message.

   b. Read - reads the next log entry for presentation to the operator.

5. Control Intransit Thresholds - allows the operator to check or change the threshold levels for Intransit memory. The value of the thresholds (in percent) determines when Overflow Storage will be activated and deactivated.

   a. Read - allows the operator to examine the current thresholds.

   b. Set - allows the operator to set or modify the current thresholds.

6. Access Message - provides facilities for monitoring, correcting, originating and terminating messages.

   a. Read - allows the operator to examine a message which is already in the system.

   b. Modify - allows the operator to change or correct a message.

   c. Generate - allows the operator to manually type in a message in one of several predefined formats.

   d. Insert - places or replaces a message in the system.

   e. Remove - takes a message out of the system.

   f. Save - causes a message to be saved in volatile storage.

7. Print - is an output driver which allows certain types of information and operator requested items to be printed.

8. Maintain Switch - contains the off-line diagnostics, and fault isolation functions as well as on-line maintenance functions and the Supervisor/Maintainer functions.

Figure 2.2-3 (continued)

9. Notify Operator - an output interface to the operator for special purpose, time critical or warning information.

10. Control Status - provides capabilities for the operator to examine and change the various items of status and condition information.

    a. Read - allows the operator to examine the current status.

    b. Modify - allows the operator to change certain status items to reflect the current conditions.

    c. Save - causes all of the current status information to be saved in nonvolatile storage.

Figure 2.2-4

O. Stop Receiving Messages - the first step in any ordered shut-down or "DRY-UP" sequence. In general, when activated will notify each receiving function to cease accepting new messages. Qualifiers may be applied to initiate a slower or a line specific dry-up.

P. Finish Transmitting Messages - this function may be initiated to cease transmission on any or all lines either when transmission of the current message is complete or when all of the queued messages have been transmitted.

Q. Deactivate Channels - causes the transmission of control characters or service messages, as appropriate, to indicate the impending channel deactivation. After this transmission channels are removed from active service.

R. Save All Messages Not Transmitted - messages or message versions which have not been transmitted are saved in nonvolatile storage, for later recovery and transmission.

S. HISTORY_OPS.CLOSE_HISTORY - A standard package that closes the day's history and saves the current audit. The day's history is reaudited and the generated audit is compared to the saved running audit. Statistics are compiled for the day.

T. Notify Operator - at the various stages of a normal stop the operator will be advised of the current progress and status. The operator will also be advised after the initial operations of an emergency stop are completed.

Figure 2.2-4 (continued)

U. Terminate Input/Output Processing - the receipt of incoming messages and the transmission of outgoing messages will be terminated immediately when an emergency condition is detected.

V. Finish Writing Log Entries - log entries which are in progress or pending are completed so that the most complete journal will be available for recovery. This reduces the probability of transmission of duplicate messages.

W. Save Running Audit - the current running audit is saved in nonvolatile storage to be used as a cross-check during recovery.

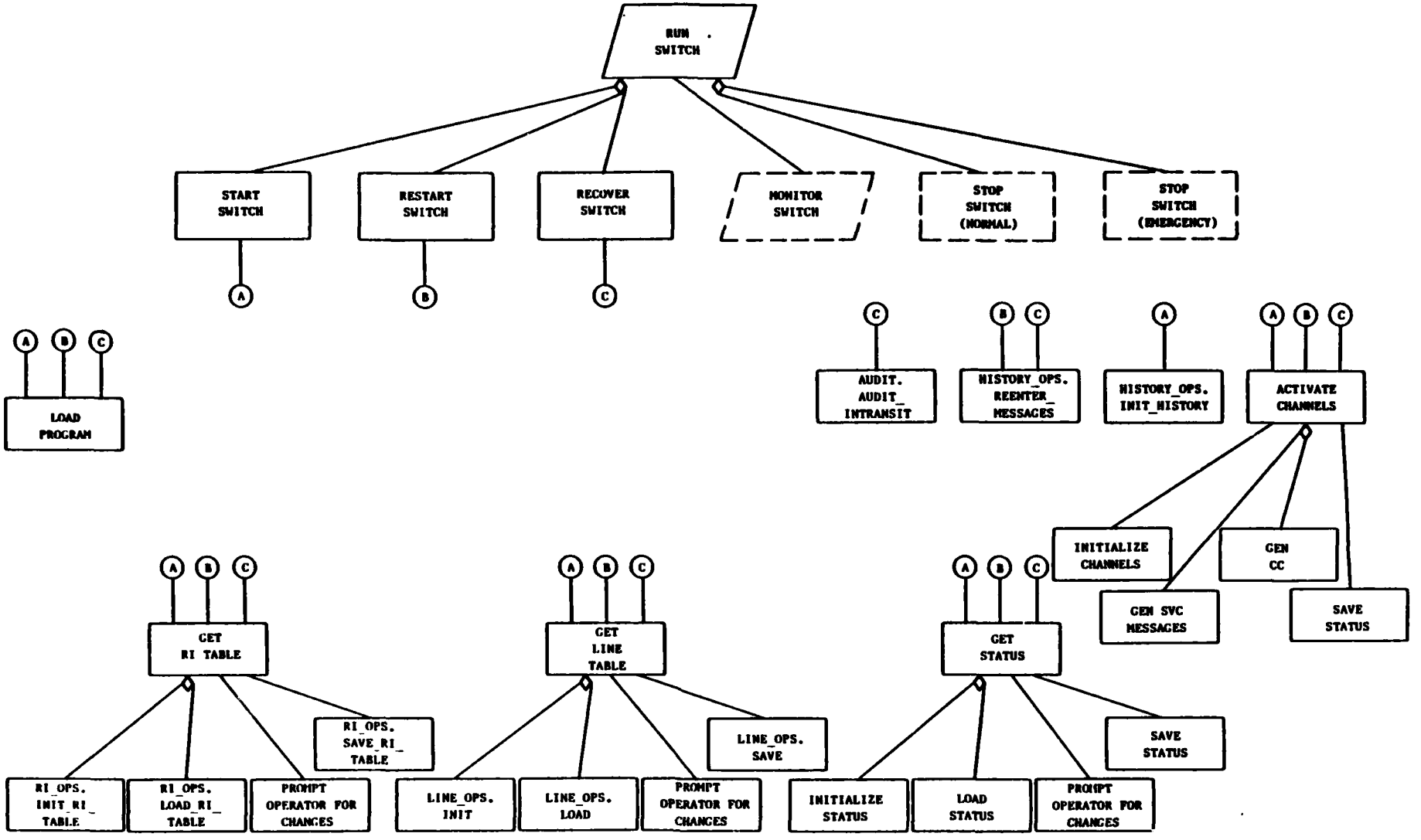



Figure 2.2-1 Run Switch

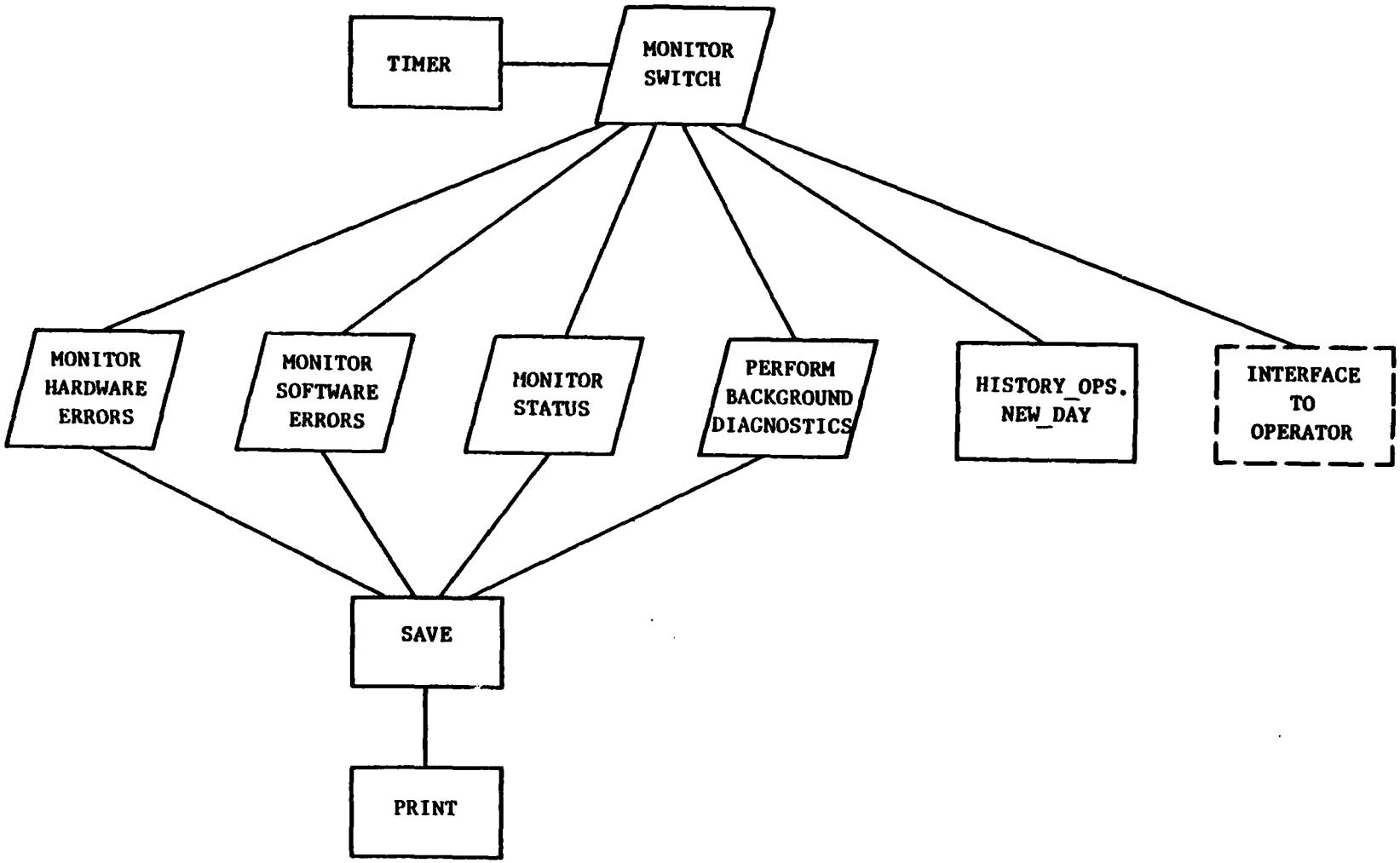



Figure 2.2-2 Run Switch

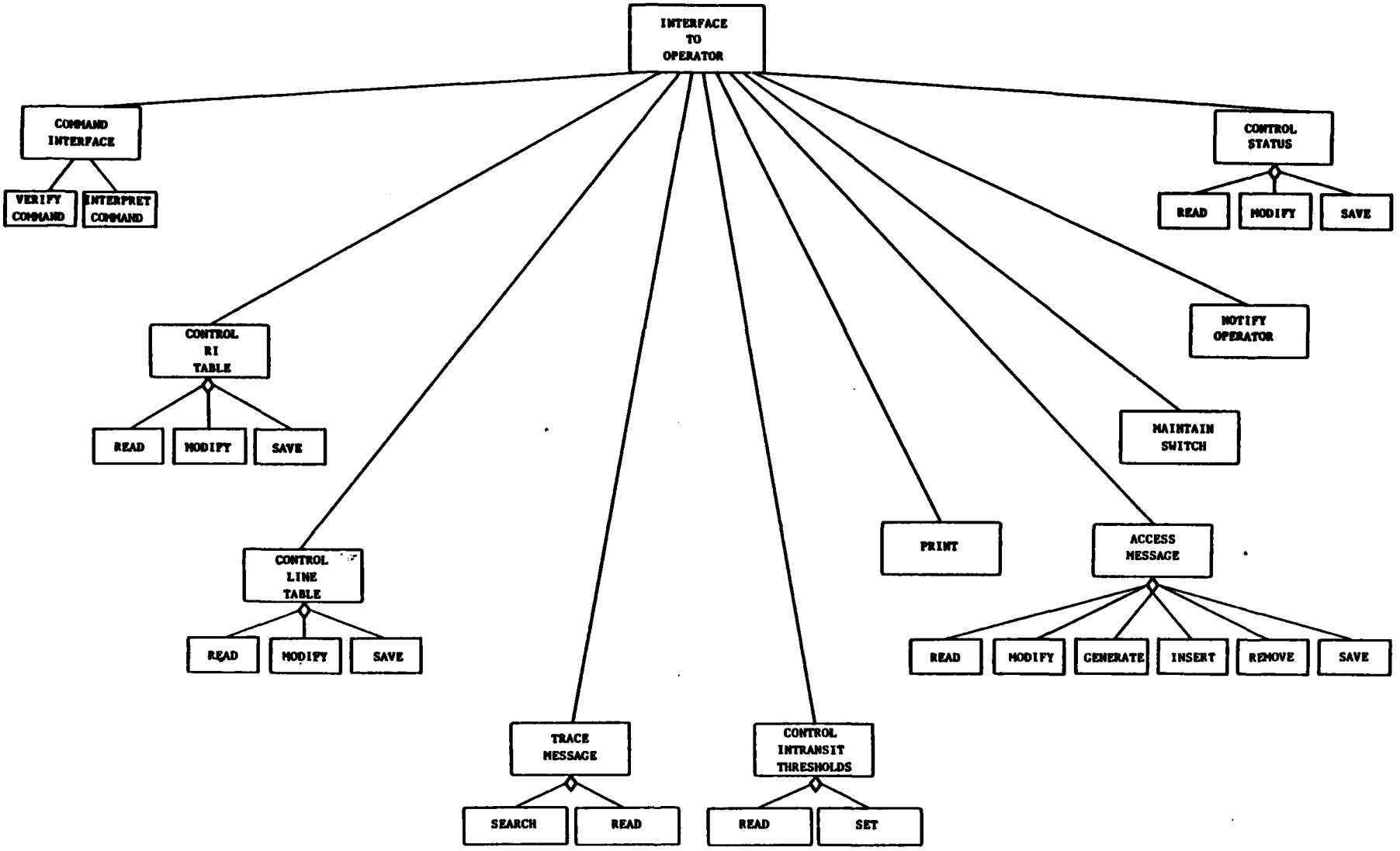



Figure 2.2-3 Run Switch

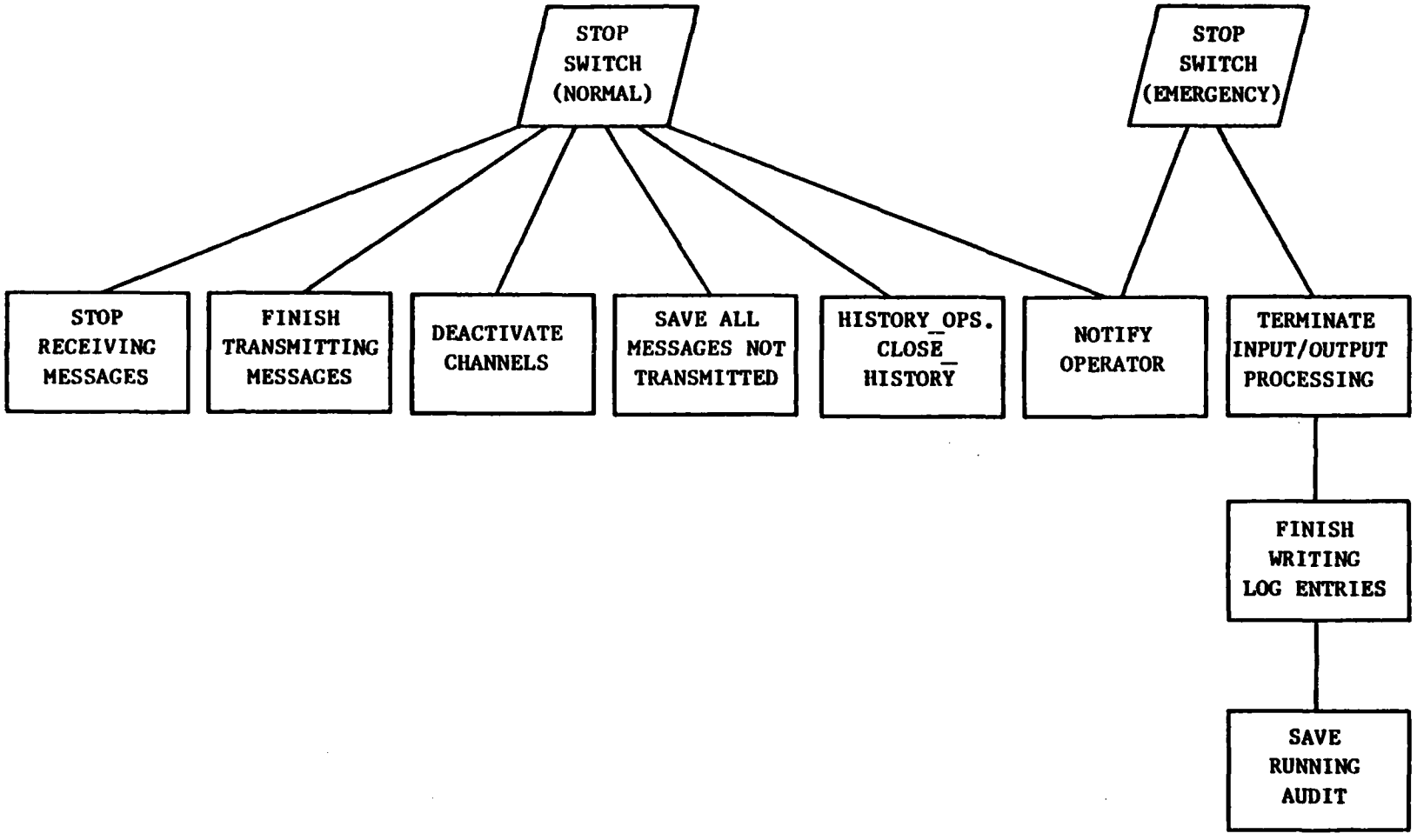



Figure 2.2-4 Run Switch

## 2.3 Internal Message Structure

The message schema shown in Figure 2.3-1 was chosen to take advantage of two facets of the messages handled by the switch.

First, the messages are received in pieces. Those messages which are sent in synchronous mode are received in blocks of 80 characters. Asynchronous messages may be interrupted by sequences of control characters. In addition to these problems, the first part of the message must be validated during the reception of the remainder, since the message must be acknowledged very shortly after the end of message is received. Therefore, the message is stored in "chunks" of 80 characters, which the designers chose to call segments. The segments are placed in a doubly linked list, using next segment and prior segment pointers. Since the segments are not stored together, information must be added to ensure that they remain properly linked together. The only item that is required by the performance specification is the security classification of the message. For further checking to be sure that no segments are lost or improperly linked, the following information was added: the serial number of the nessage, the part of the message involved (MCB, header, body, or trailer), and a number which starts with one for each part, and increases by one for each segment. The segments are also time-stamped for logging purposes.

Most messages are split into more than one copy during processing. In order to save storage, the designers decided to share text between versions of messages at the part level. To do so, a structure called the part header was created. The part headers are linked to the first and last segments of their respective parts. The part header contains the name of the part involved, a count of the number of messages which are using this part, and a count of the segments which comprise the part. This last item can be compared with the segment number of the last segment in the part.

The only remaining part of the message structure is the version header. Ths information contained in the version header includes the serial number of the message plus an extension to differentiate between version, the security classification of the message, the precedence of the message, its time of receipt, its category (normal or service) the output line to which the message is routed, and the character set (ASCII or ITA) of the message. The version headers are linked to the part headers through an array of access variables indexed by the part name.

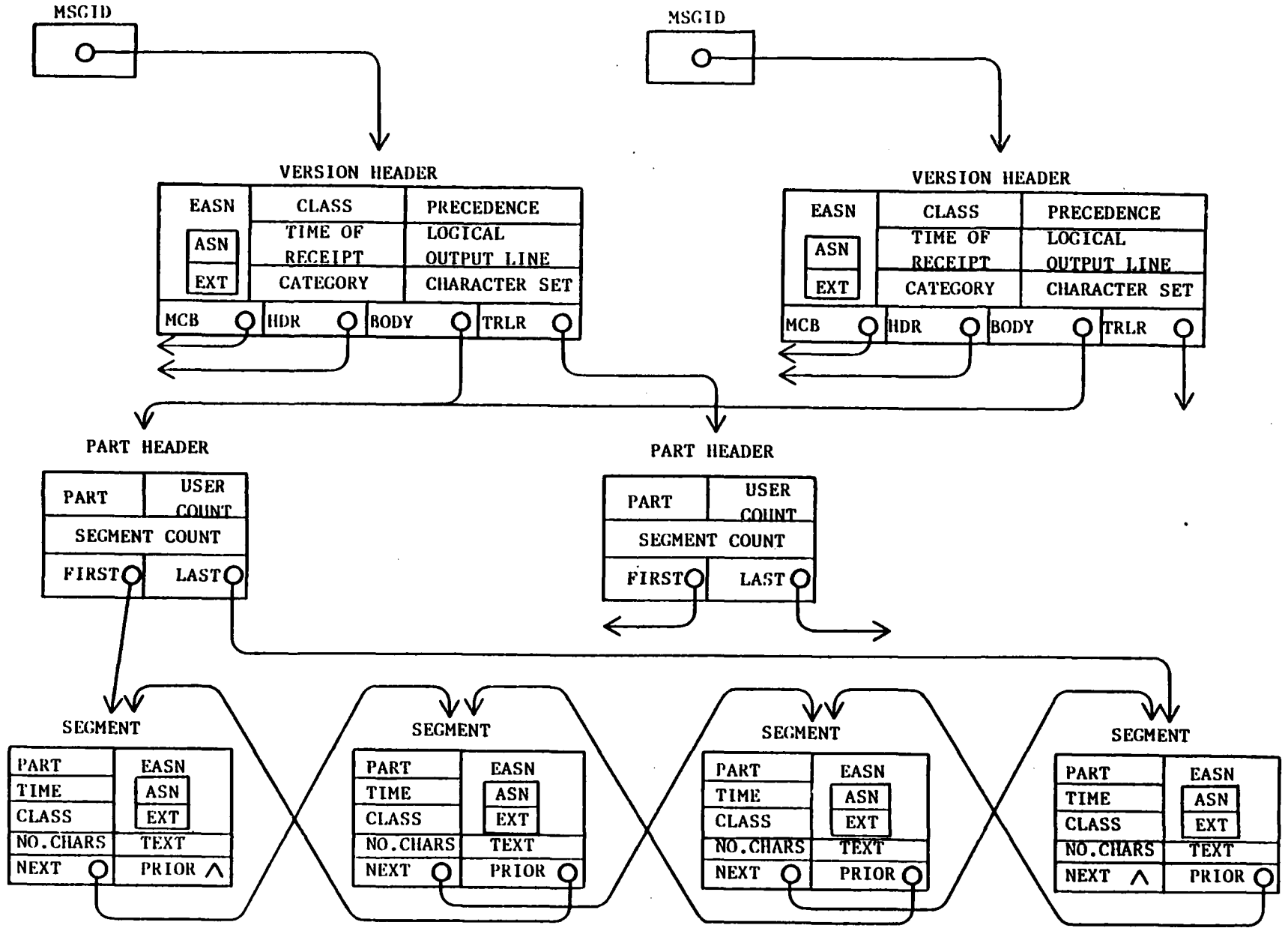



Figure 2.3-1 Message Schema

## 2.4 Functional Decomposition and Interconnectivity

### 2.4.1 Message Input

The Message Input function is illustrated in Figures 2.4-1 through 2.4-19. The dotted parallelograms indicate that the referenced task or procedure is elaborated on a separate page. Other symbology and guidelines to interpreting the charts is discussed on page 64 of the Ada Integrated Methodology.

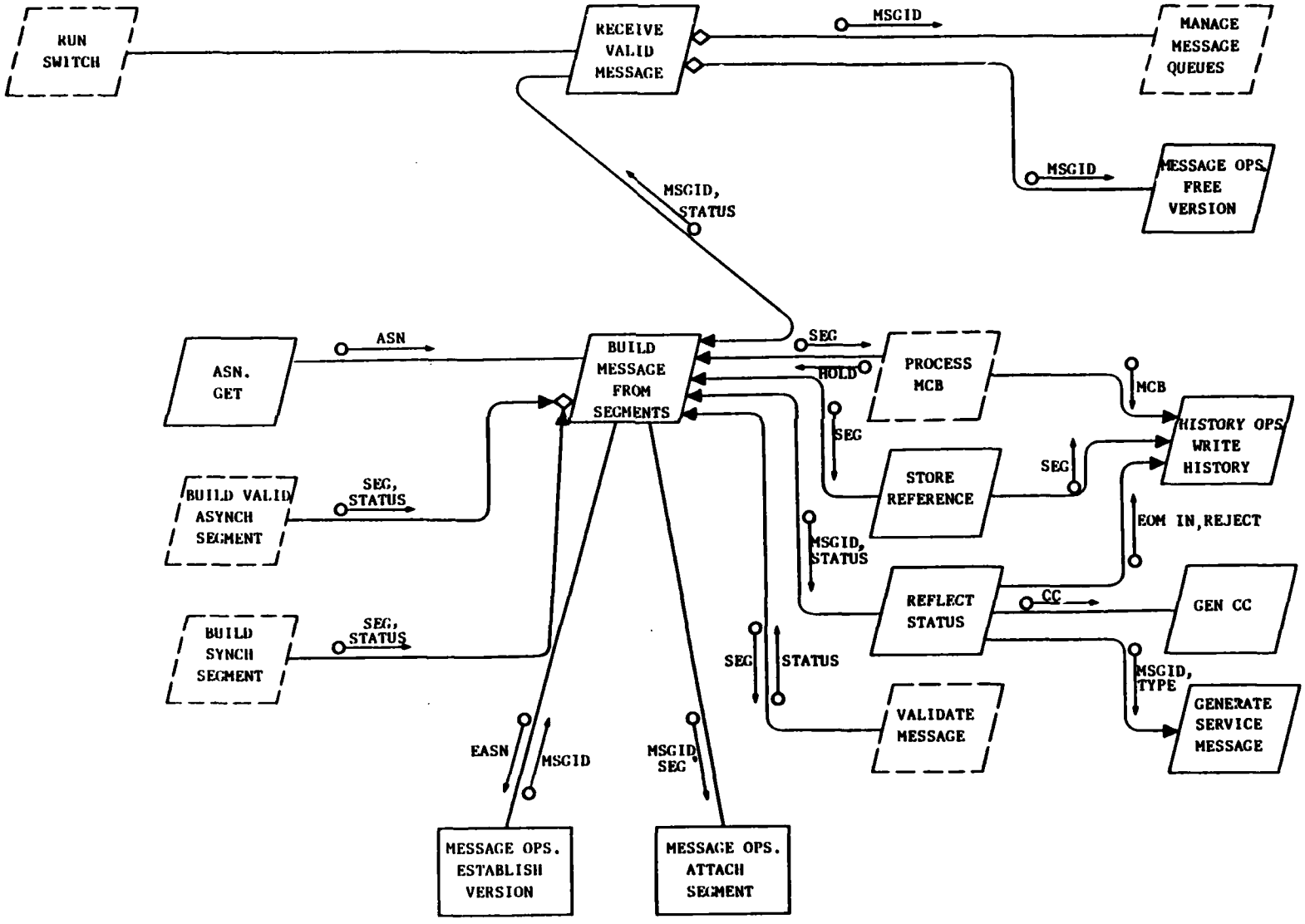



Figure 2.4-1 Receive Valid Message

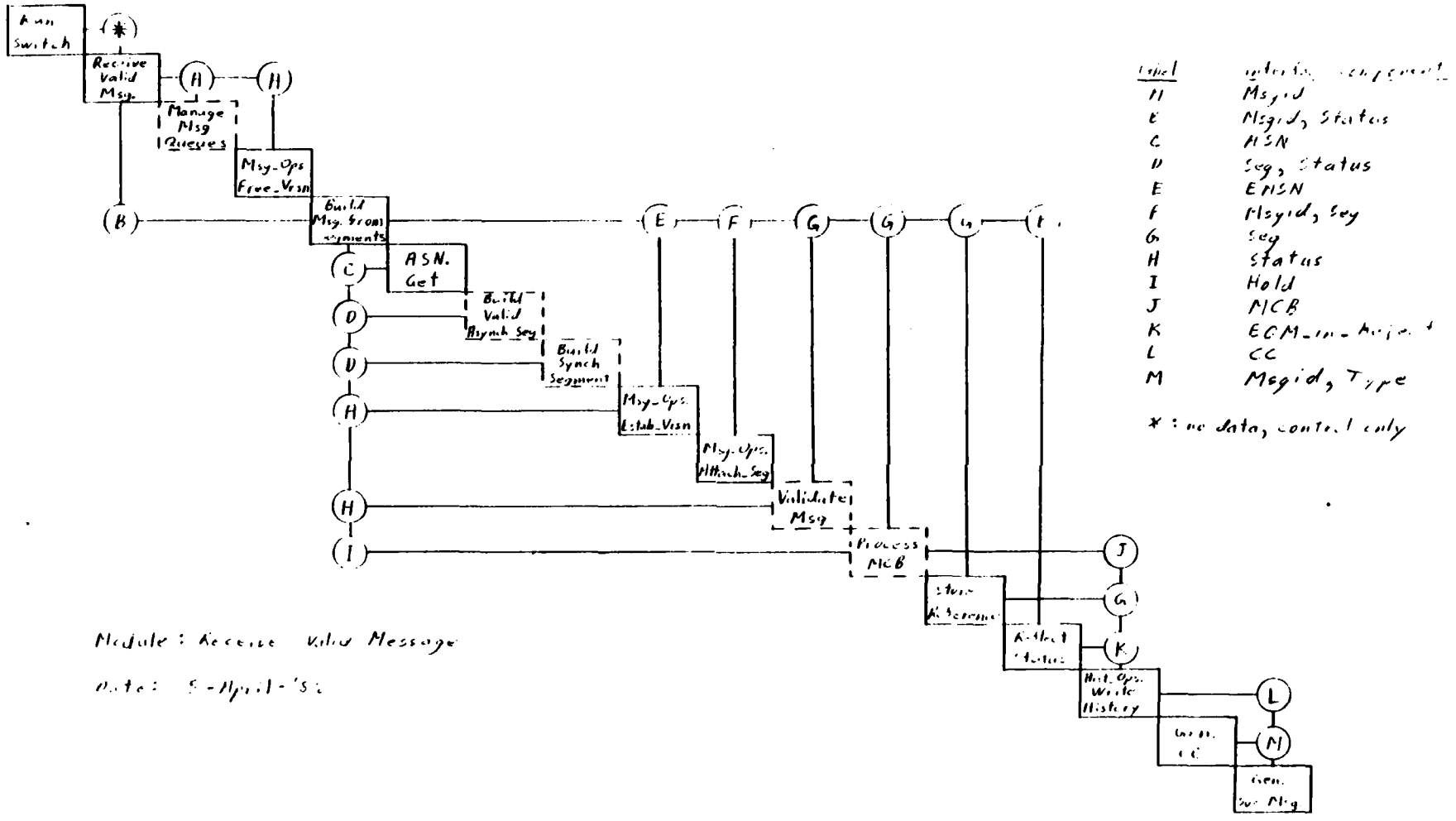



Figure 2.4-2 Receive Valid Message Interconnectivity

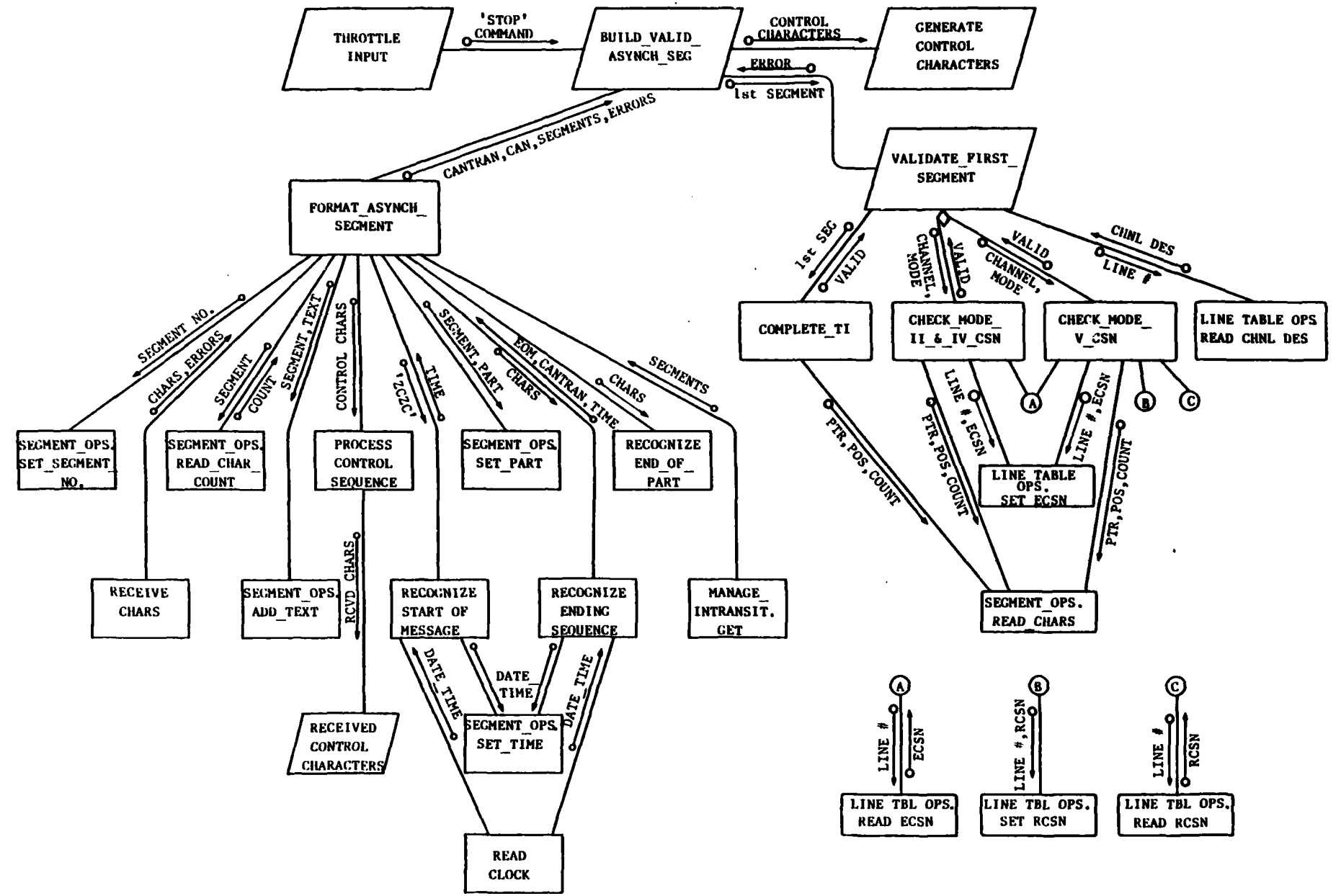
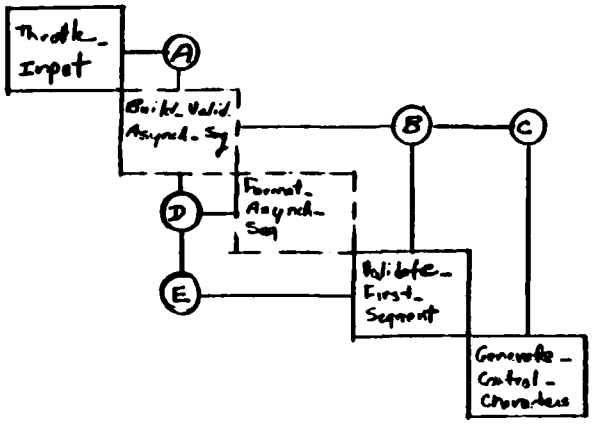


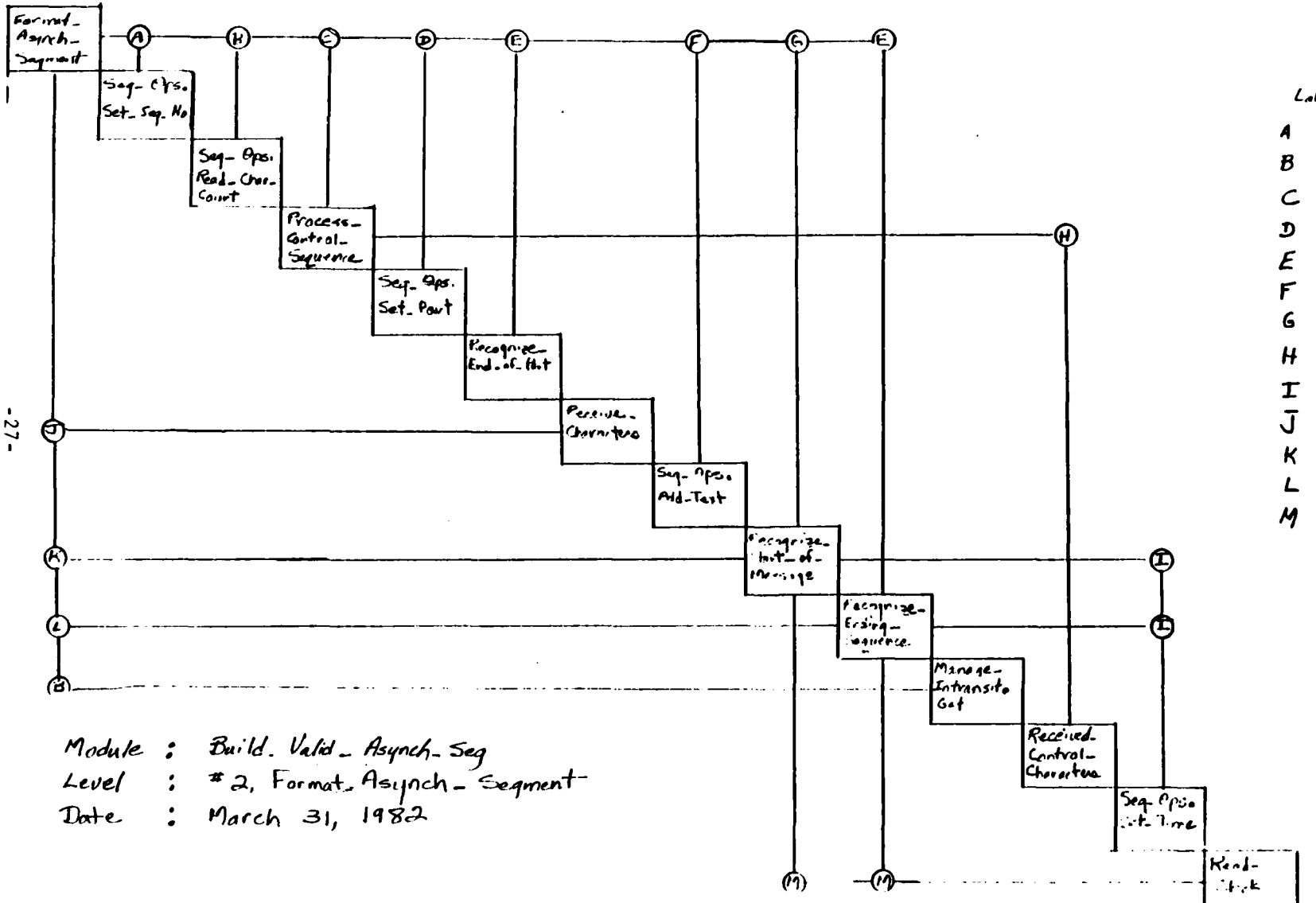

Figure 2.4-3 Build Valid Asynchronous Segment




| Label | Data Item(s) |
|---|---|
| A | 'Stop' Command |
| B | First Segment Msg ID |
| C | Control Characters |
| D | CANTRAN CAN Segment Error |
| E | Error |

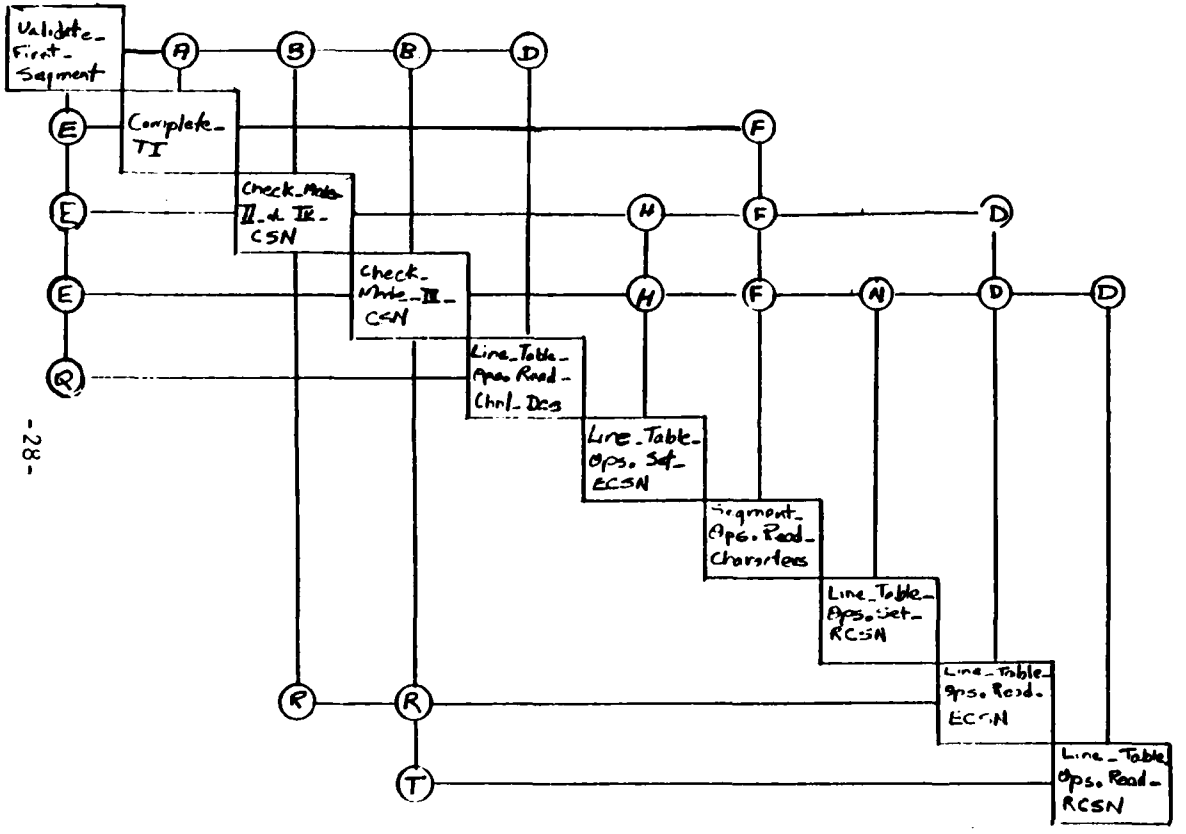
Module : Build_Valid_Asynch_Seq
Level : Top
Date : March 30, 1982

Figure 2.4-4 Build Valid Asynchronous Segment Interconnectivity




| Label | Data |
|---|---|
| A | Segment No |
| B | Segment |
| C | Control Chars |
| D | Segment, Part |
| E | Characters |
| F | Segment, Text |
| G | 'ZCZC' |
| H | Received Chars |
| I | Date, Time |
| J | Characters, Errors |
| K | Time |
| L | EOM, Center Time |
| M | Date, time |

Figure 2.4-5 Build Valid Asynchronous Segment Interconnectivity




| Label | Data Items |
|---|---|
| A | 1st Segment |
| B | Channel Mode |
| D | Line # |
| E | Valid |
| F | Pointer, Position Count |
| H | Line #, ESCN |
| N | Line #, RSCN |
| Q | Channel Desc. |
| R | ESCN |
| T | RSCN |

Module : Build_Valid_Asynch_Seg

Level : # 2, Validate First Segment

Date : March 30, 1982

Figure 2.4-6 Build Valid Asynchronous Segment Interconnectivity

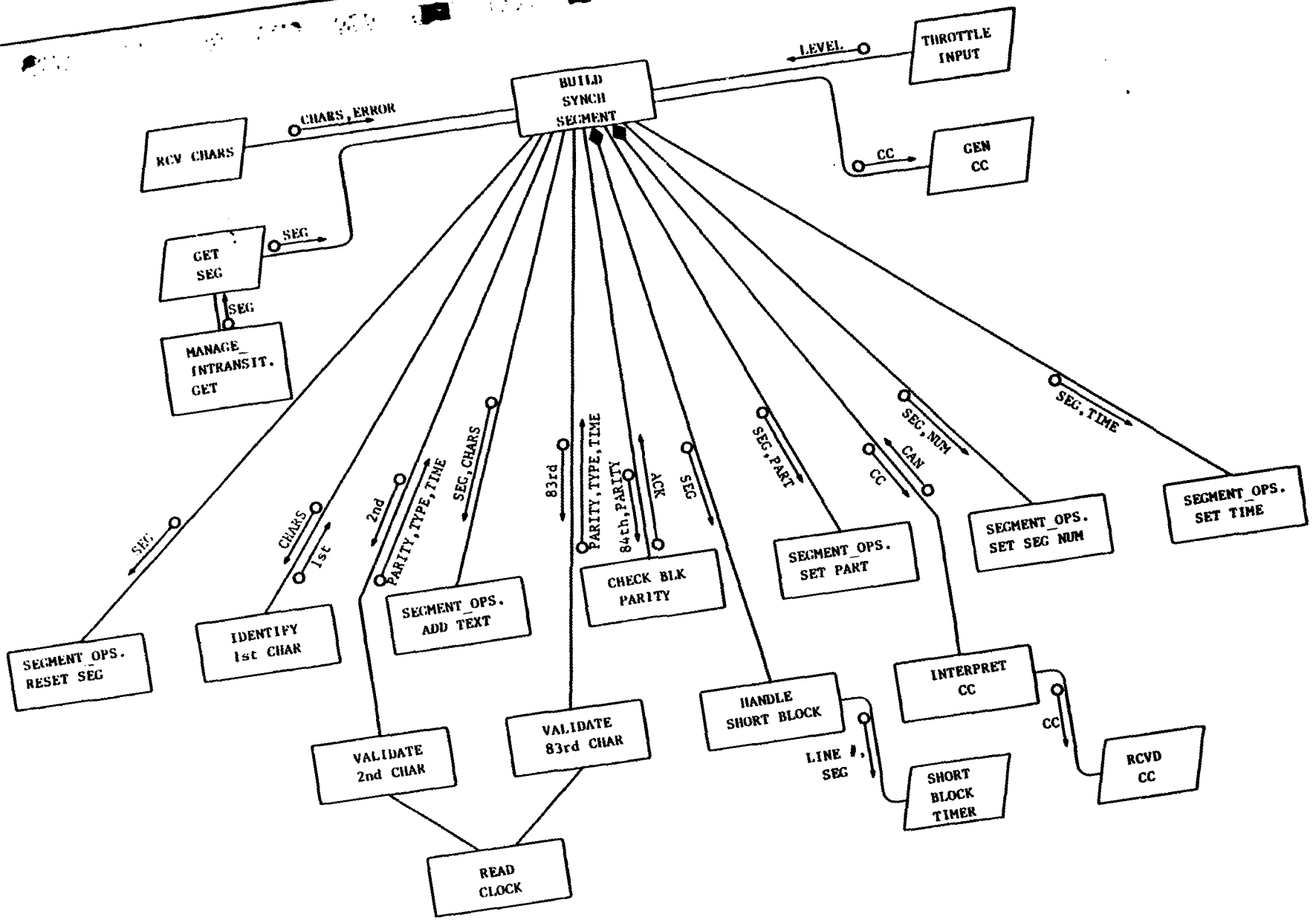
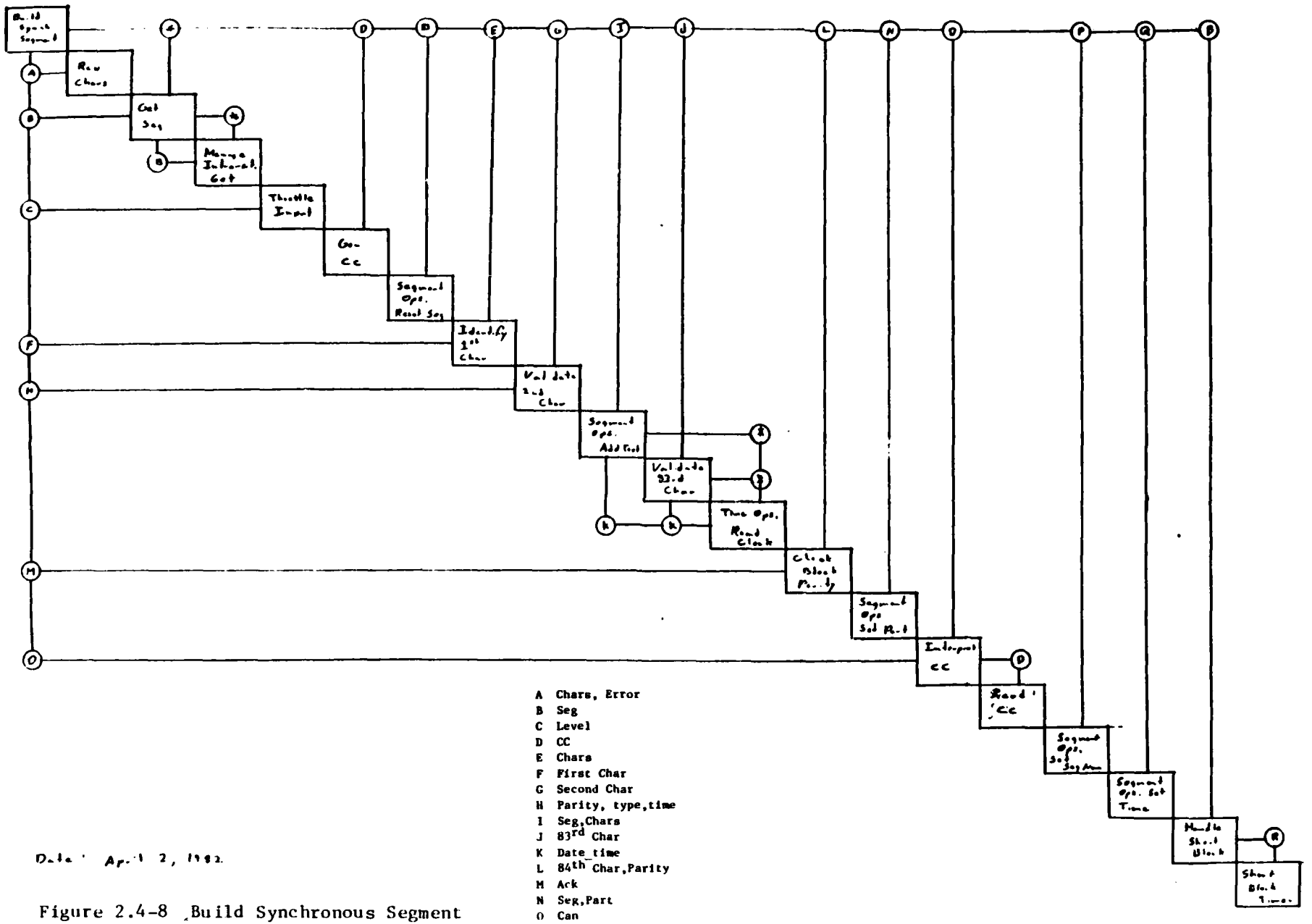



Figure 2.4-7 Build Synchronous Segment



A Chars, Error
B Seg
C Level
D CC
E Chars
F First Char
G Second Char
H Parity, type, time
I Seg, Chars
J 83rd Char
K Date_time
L 84th Char, Parity
M Ack
N Seg, Part
O Can
P Seg, Num
Q Seg, Time
R Seg, Line #
* No data, Call only

Date: April 2, 1982

Figure 2.4-8 Build Synchronous Segment Interconnectivity

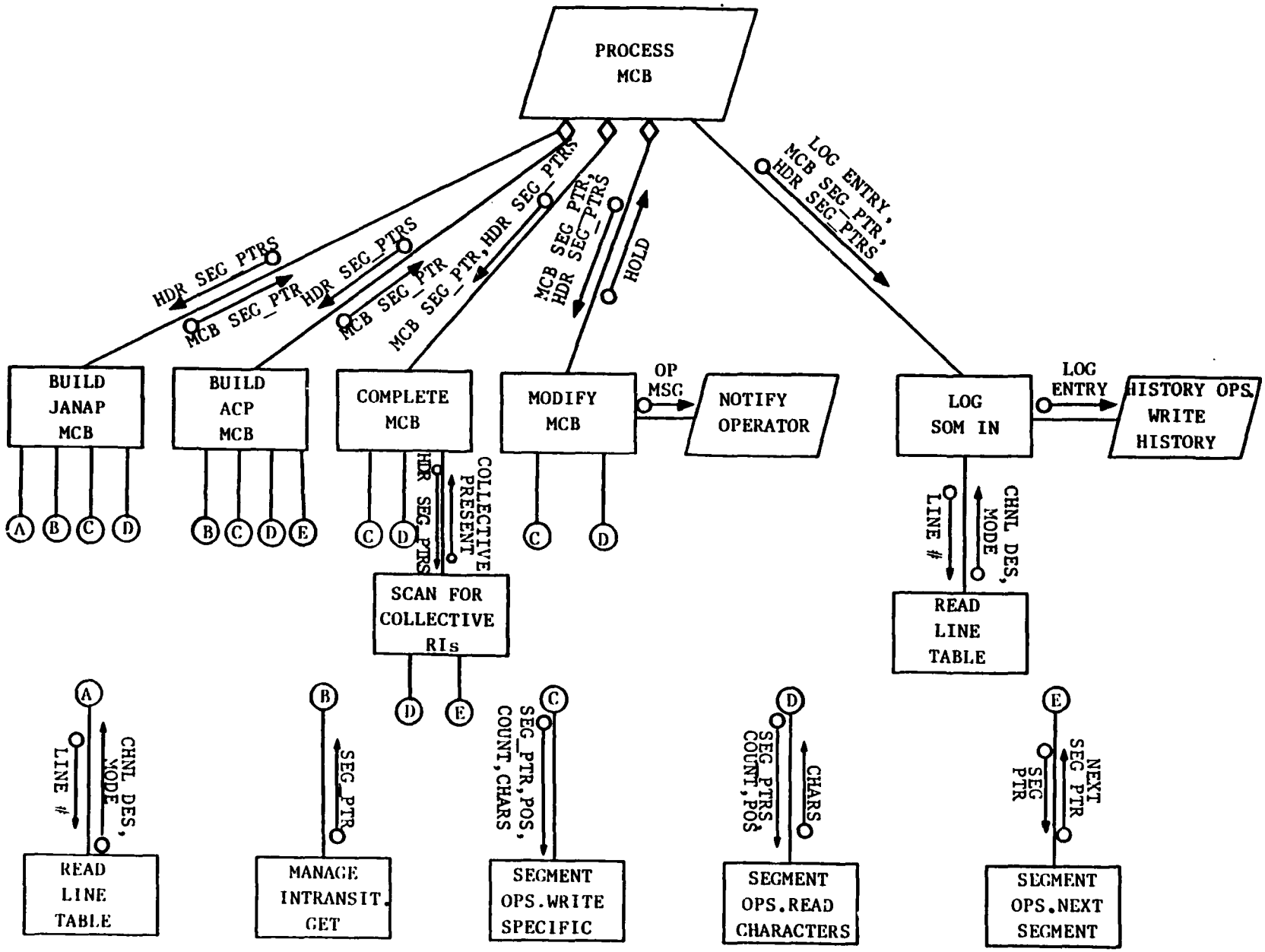



Figure 2.4-9 Process MCB

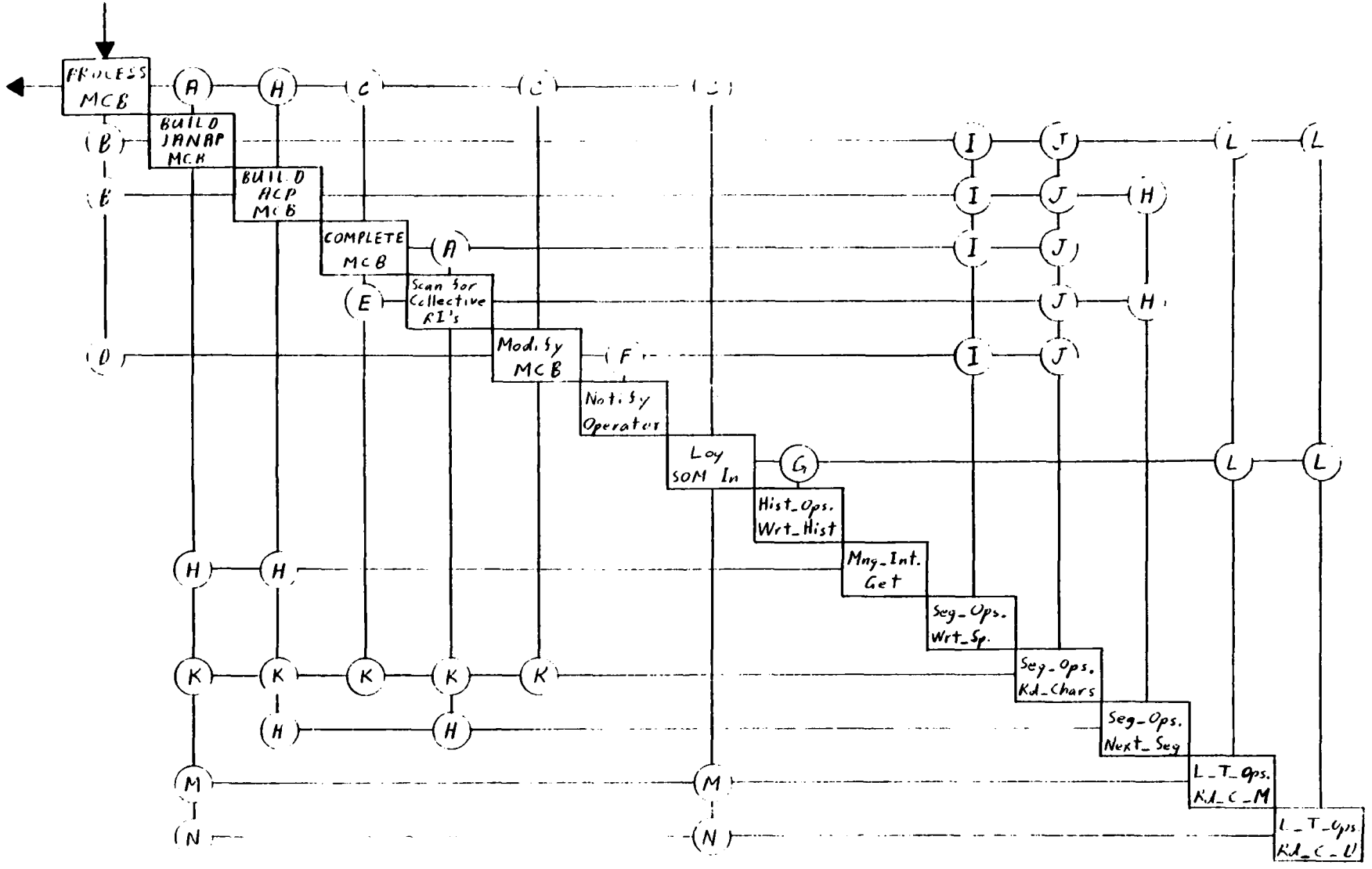
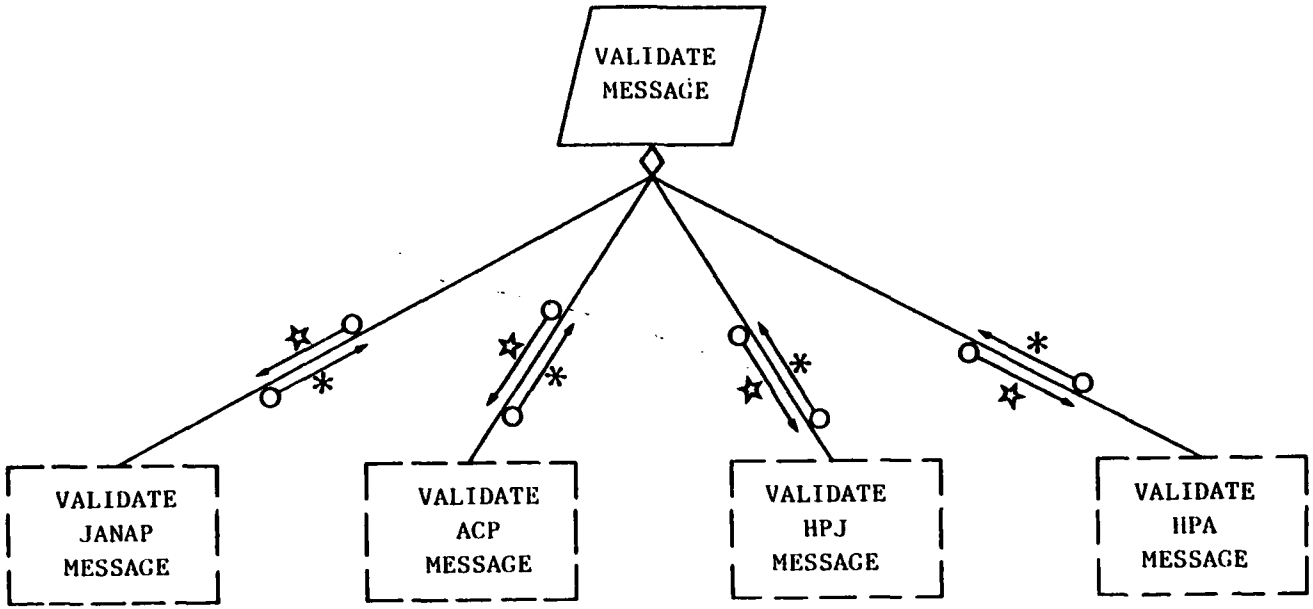



Figure 2.4-10 Process MCB Interconnectivity

| LABEL | INTERFACE COMPONENTS |
|---|---|
| A | Hdr_Seg_Ptrs |
| B | MCB_Seg_Ptr |
| C | MCB_Seg_Ptr,Hdr_Seg_Ptrs |
| D | Hold |
| E | Collective_Present |
| F | Operator_Message |
| G | Log_Entry |
| H | Seg_Ptr |
| I | Seg_Ptr,Position,Count,Chars |
| J | Seg_Ptr,Count,Position |
| K | Chars |
| L | Line # |
| M | Channel_Mode |
| N | Channel_Des |

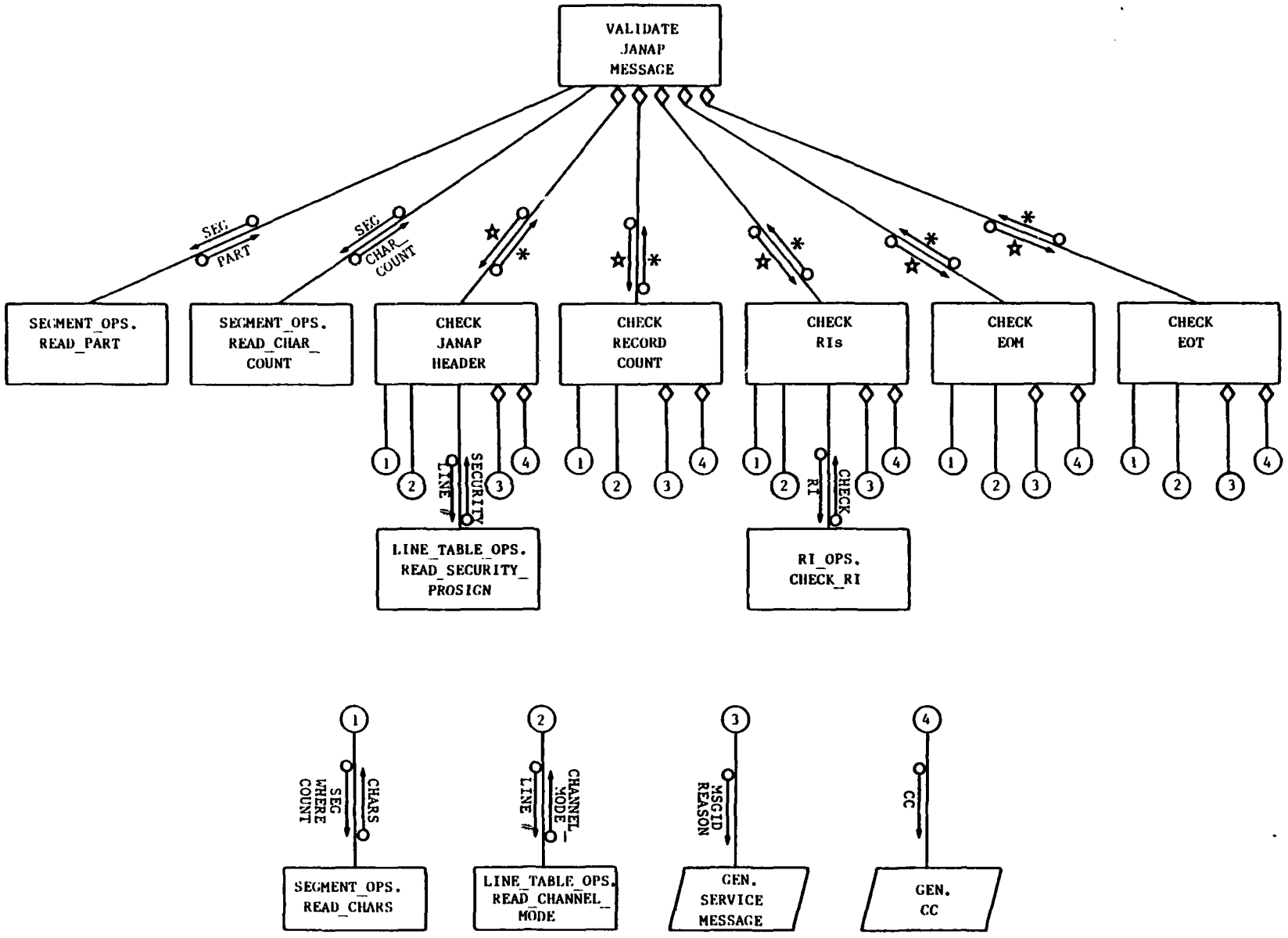
Figure 2.4-11 Chart for Process MCB






☆ = STATE,MSGID,SEG
✱ = STATE

Figure 2.4-12 Validate Message




☆ = MSGID, SEG, CHAR COUNT, WHERE, STATE

* - WHERE, STATE

Figure 2.4-13 Validate JANAP Message

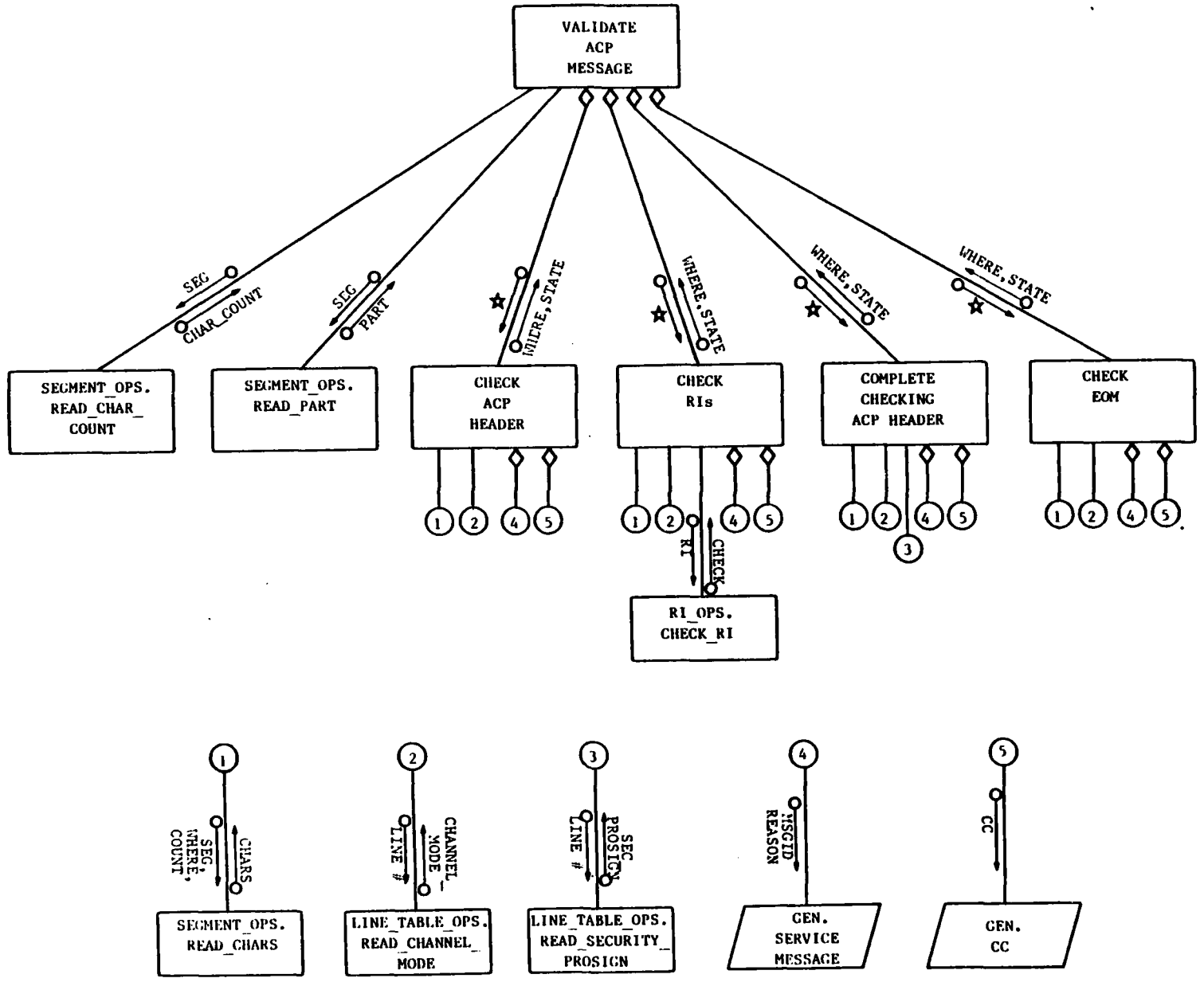
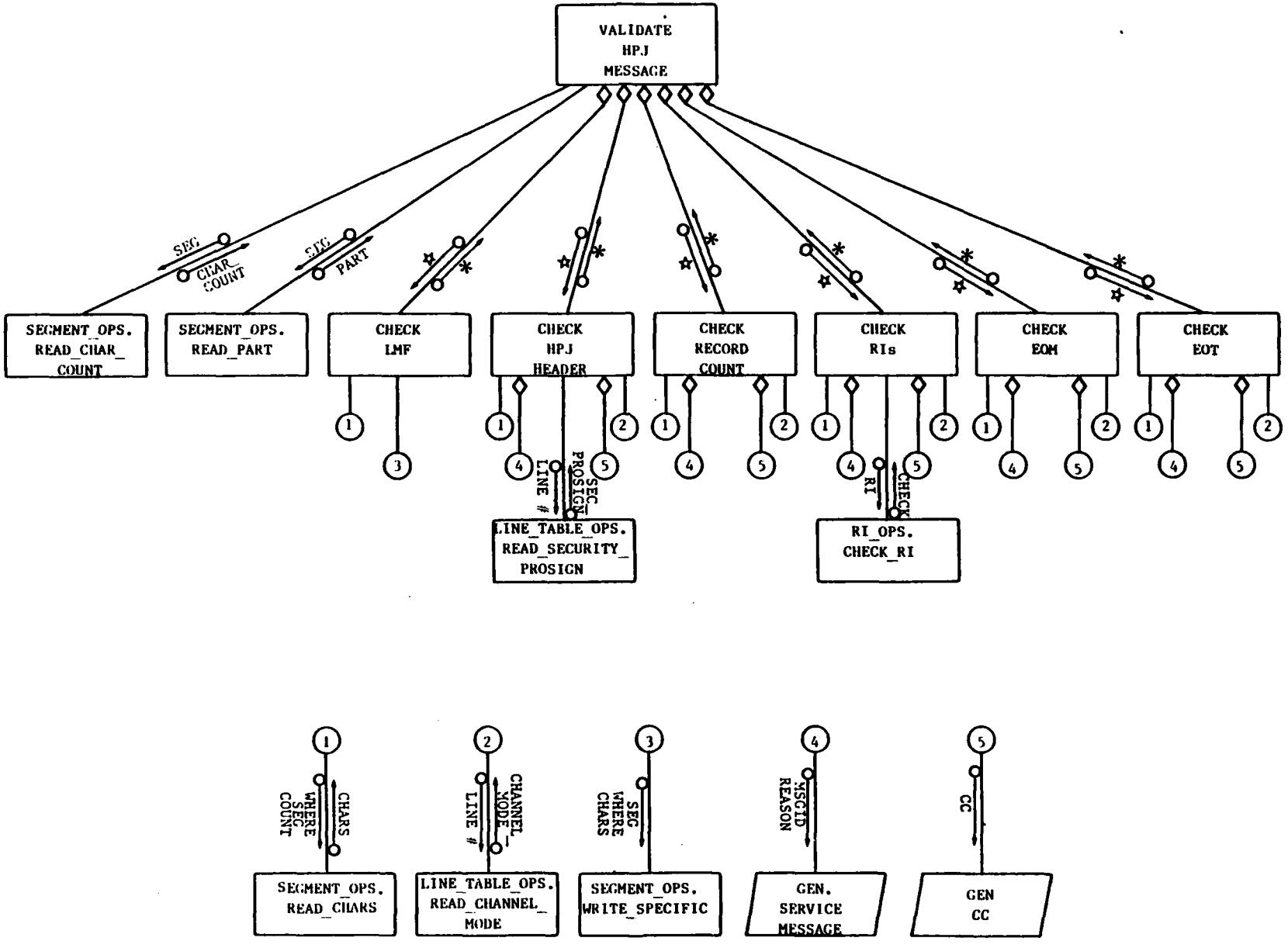



Figure 2.4-14 Validate ACP Message




☆ = MSGID,SEG,CHAR_COUNT,WHERE,STATE
✱ = WHERE,STATE

Figure 2.4-15 Validate HPJ Message

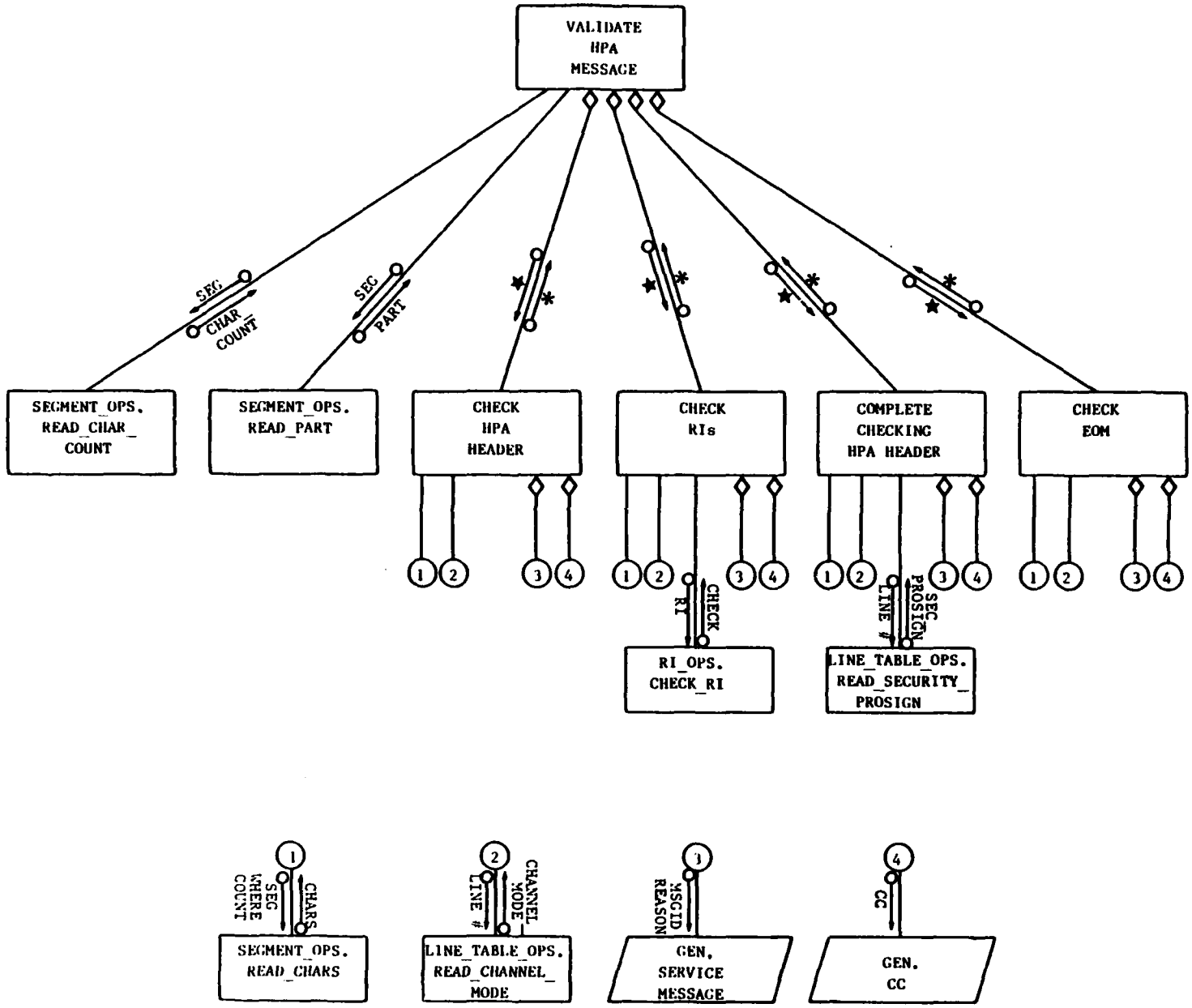
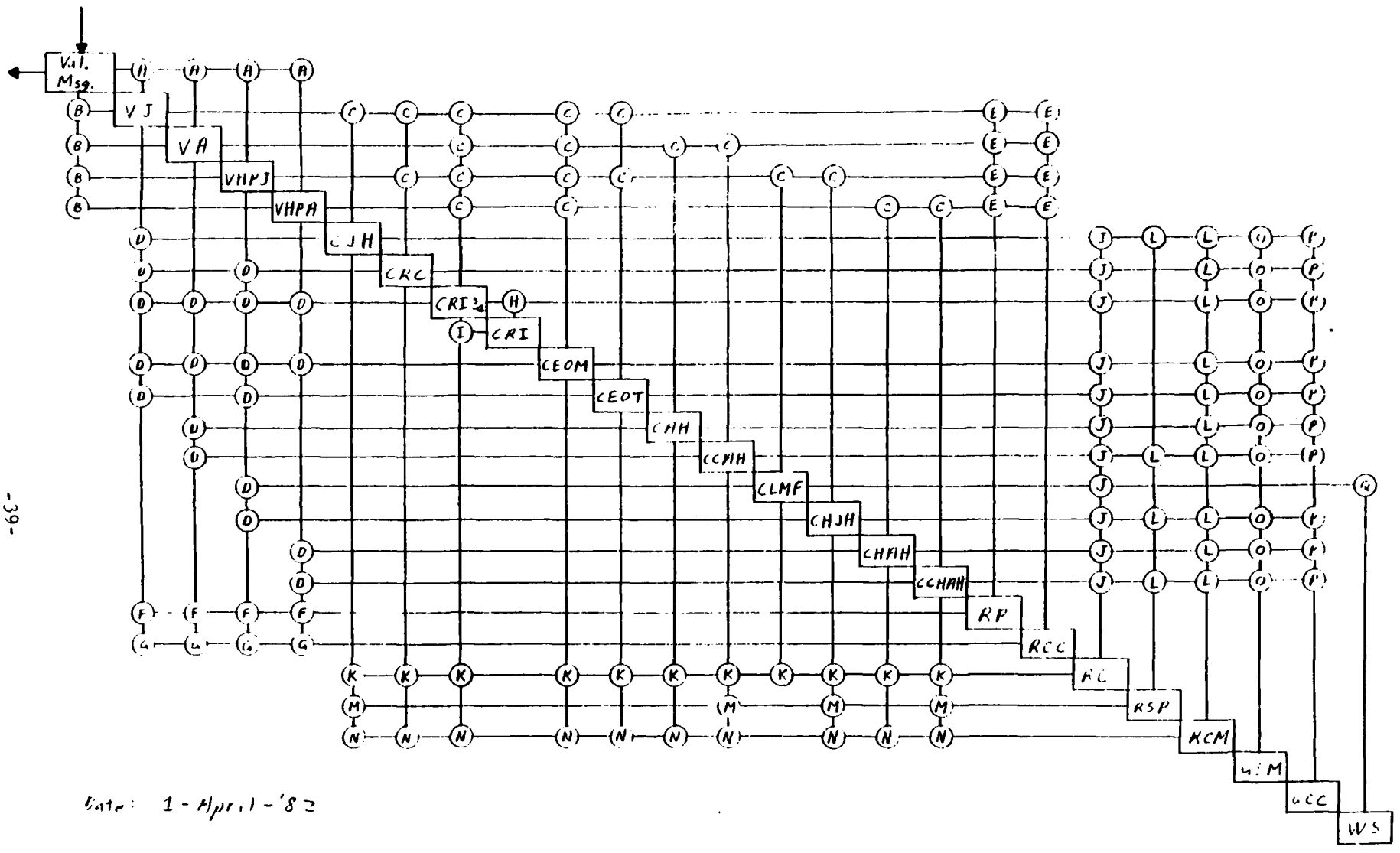



Figure 2.4-16 Validate HPA Message




Date: 1-April-'82



Figure 2.4-17 Validate Message Interconnectivity

'VALIDATE MESSAGE'

| ABBREVIATION | NAME |
|---|---|
| Val. Msg. | Validate Message |
| VJ | Validate JANAP Msg |
| VA | Validate ACP Msg |
| VHPJ | Validate HPJ Msg |
| VHPA | Validate HPA Msg |
| CJH | Check JANAP Header |
| CRC | Check Record Count |
| CRI's | Check RI's |
| CRI | RI_Ops. Check_RI |
| CEOM | Check EOM |
| CEOT | Check EOT |
| CAH | Check ACP Header |
| CCAH | Complete Checking ACP Header |
| CLMF | Check LMF |
| CHSH | Check HPJ Header |
| CHAH | Check HPA Header |
| CCHAH | Complete Checking HPA Header |
| RP | Segment_Ops.Read_Part |
| RCC | Segment_Ops.Read_Char_Count |
| RC | Segment_Ops.Read_Chars |
| RSP | Line_Table_Ops.Read_Sec_Prosign |
| RCM | Line_Table_Ops.Read_Channel_Mode |
| GSM | Generate Service Message |
| GCC | Generate CC |
| WS | Segment_Ops.Write_Specific |

Figure 2.4-18 Chart for Validate Message

'Reveive_Valid_Message'
'Build_Message_from_Segments'
'Validate_Message'

| LABEL | INTERFACE COMPONENTS |
|---|---|
| A | State,Msgid,Seg |
| B | State |
| C | Msgid,Seg,Char_count,Where,State |
| D | Where,State |
| E | Seg |
| F | Part |
| G | Char_count |
| H | RI |
| I | Check |
| J | Seg,Where,Count |
| K | Chars |
| L | Line # |
| M | Security_Prosign |
| N | Channel_Mode |
| O | Msgid,Reason_for_Svc_Msg |
| P | Msgid,Reason_for_CC |
| Q | Seg,Where,Chars |

Figure 2.4-19 Chart for Receive Valid Message
Build Message from Segments
Validate Message

### 2.4.2 Message Queueing

The Message Queueing function is illustrated in Figures 2.4-20 through 2.4-23.

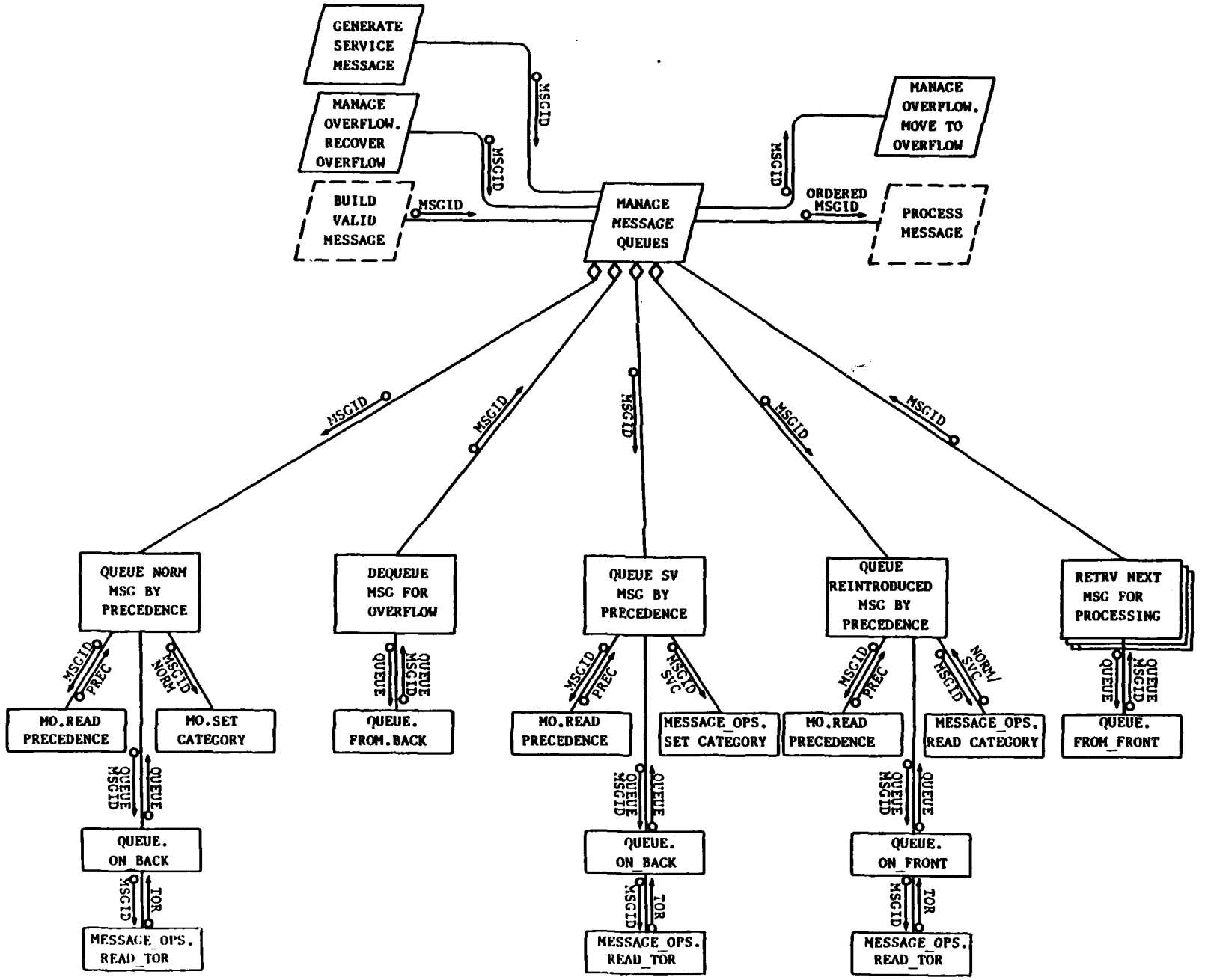



Figure 2.4-20 Manage Message Queues

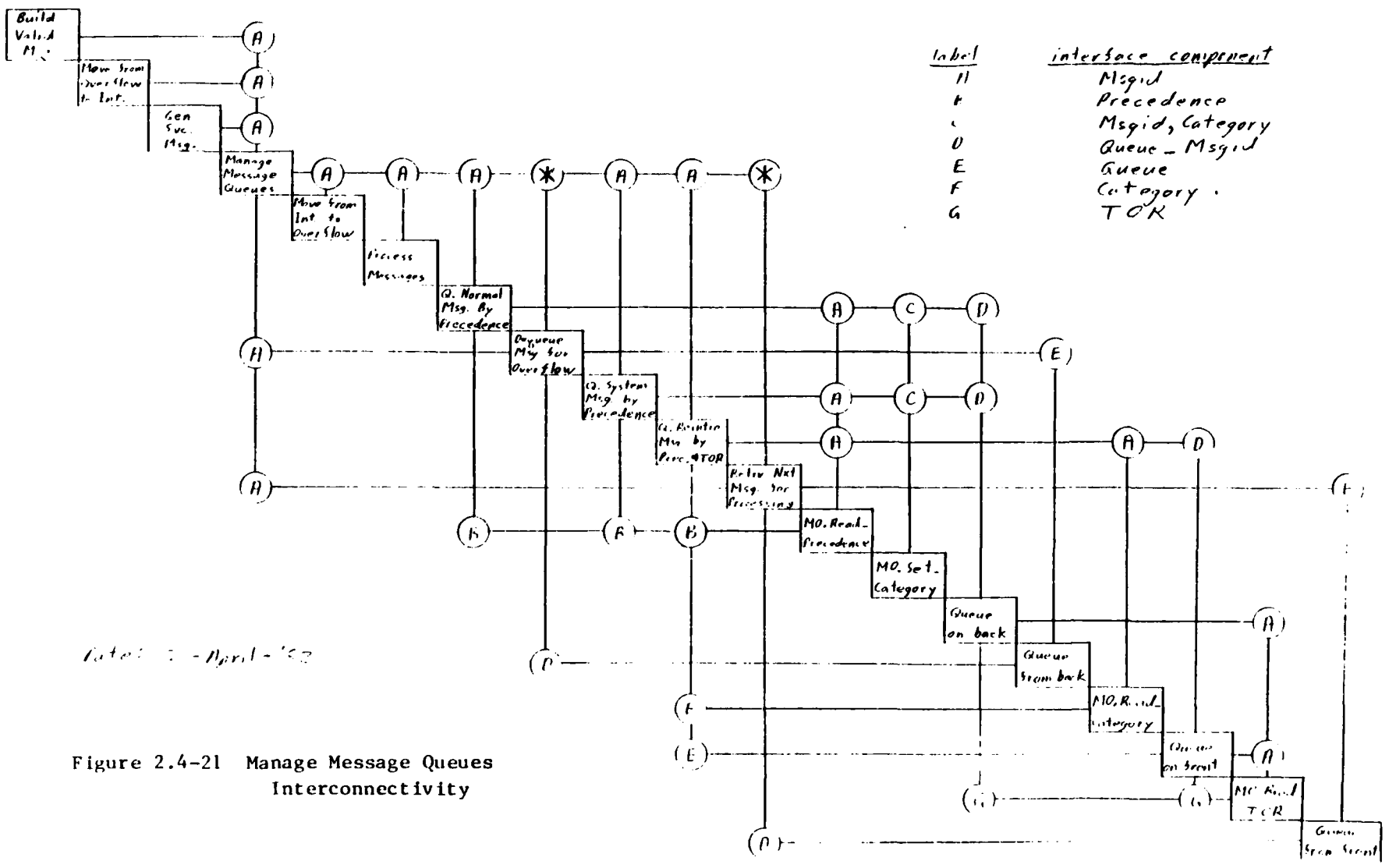



Figure 2.4-21 Manage Message Queues Interconnectivity

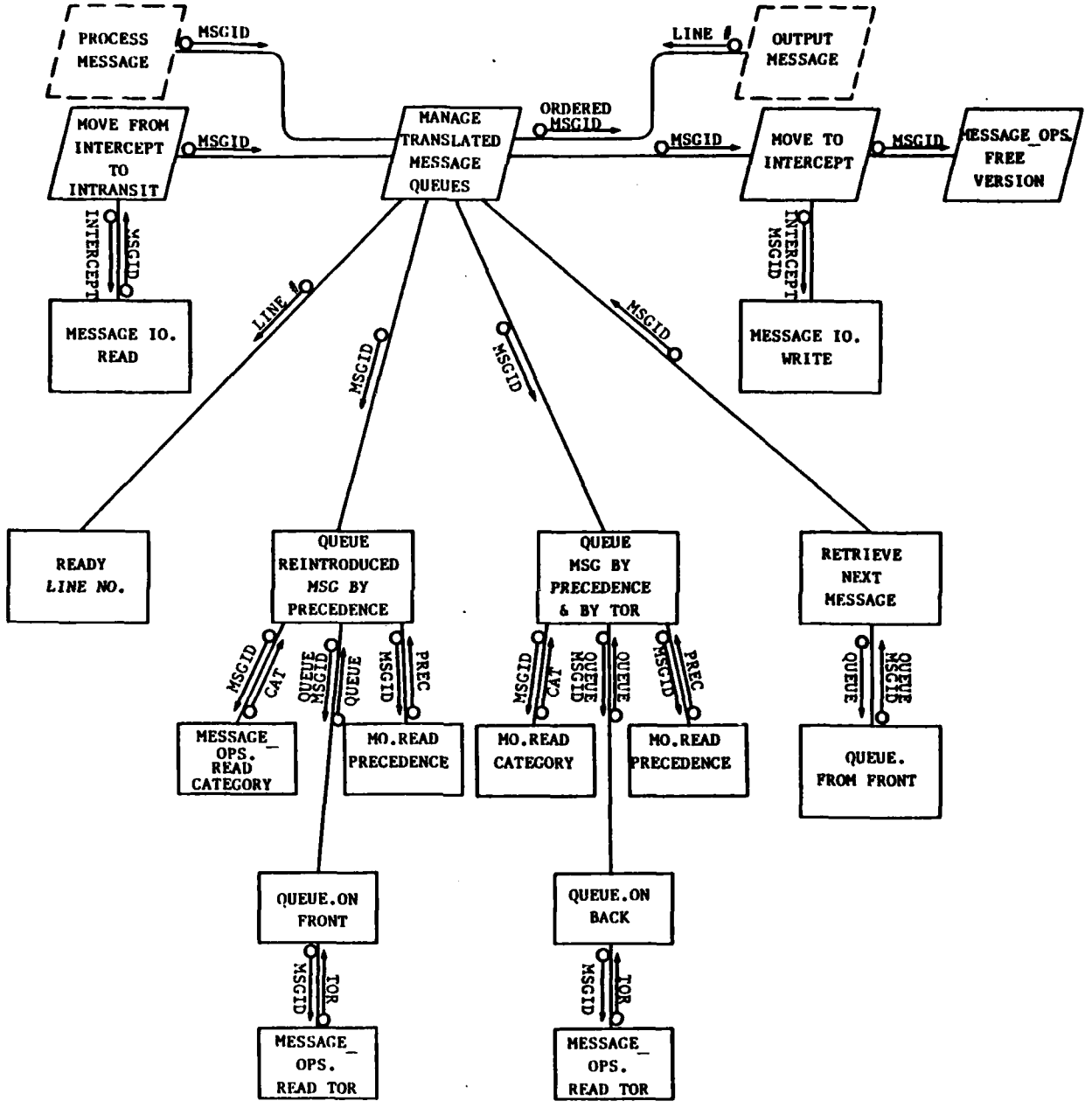



Figure 2.4-22 Manage Translated Message Queues

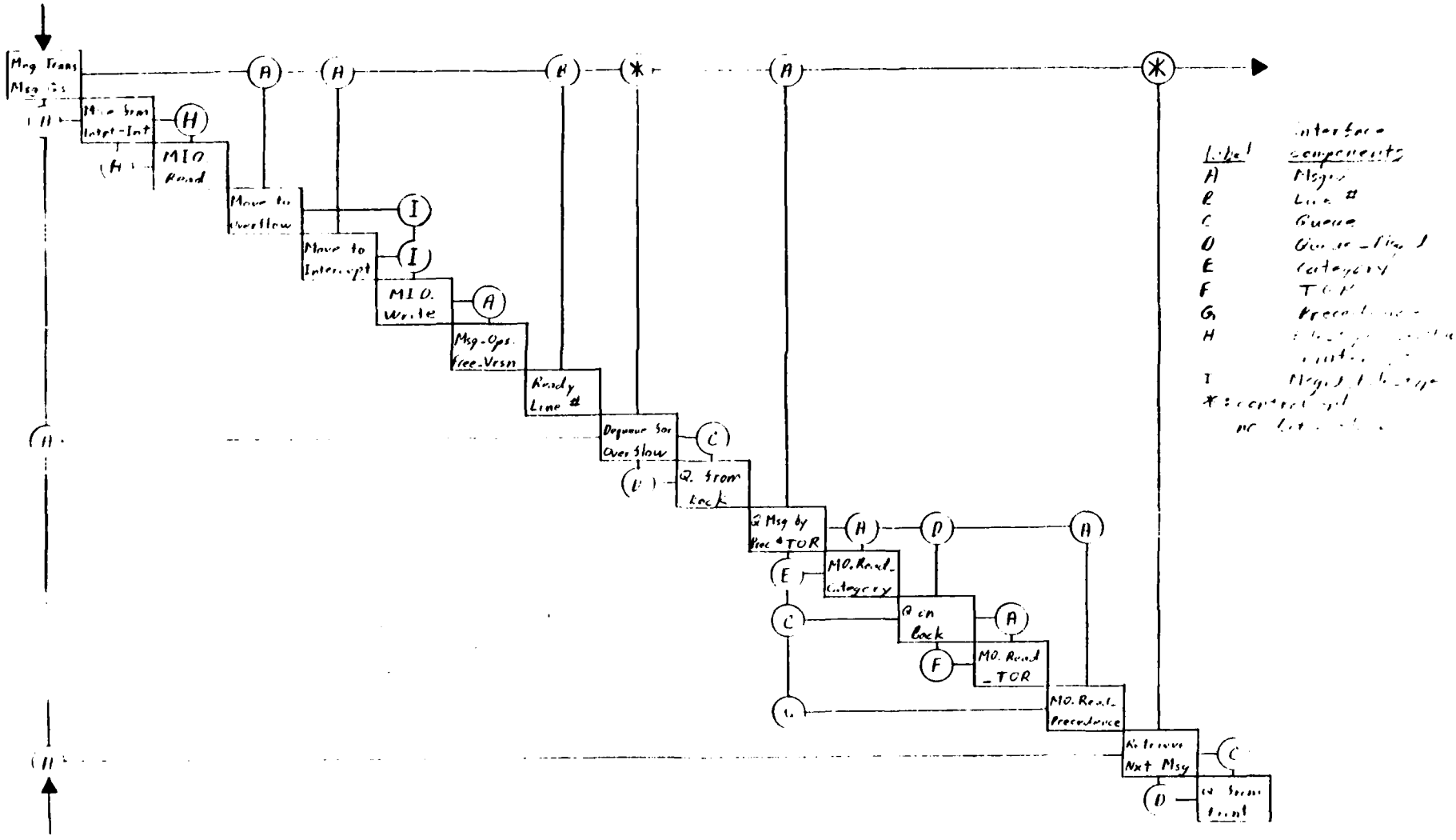



Figure 2.4-23 Manage Translated Message Queues Interconnectivity

### 2.4.3 Message Routing

The Message Routing function is illustrated in Figures 2.4-24 through 2.4-31.

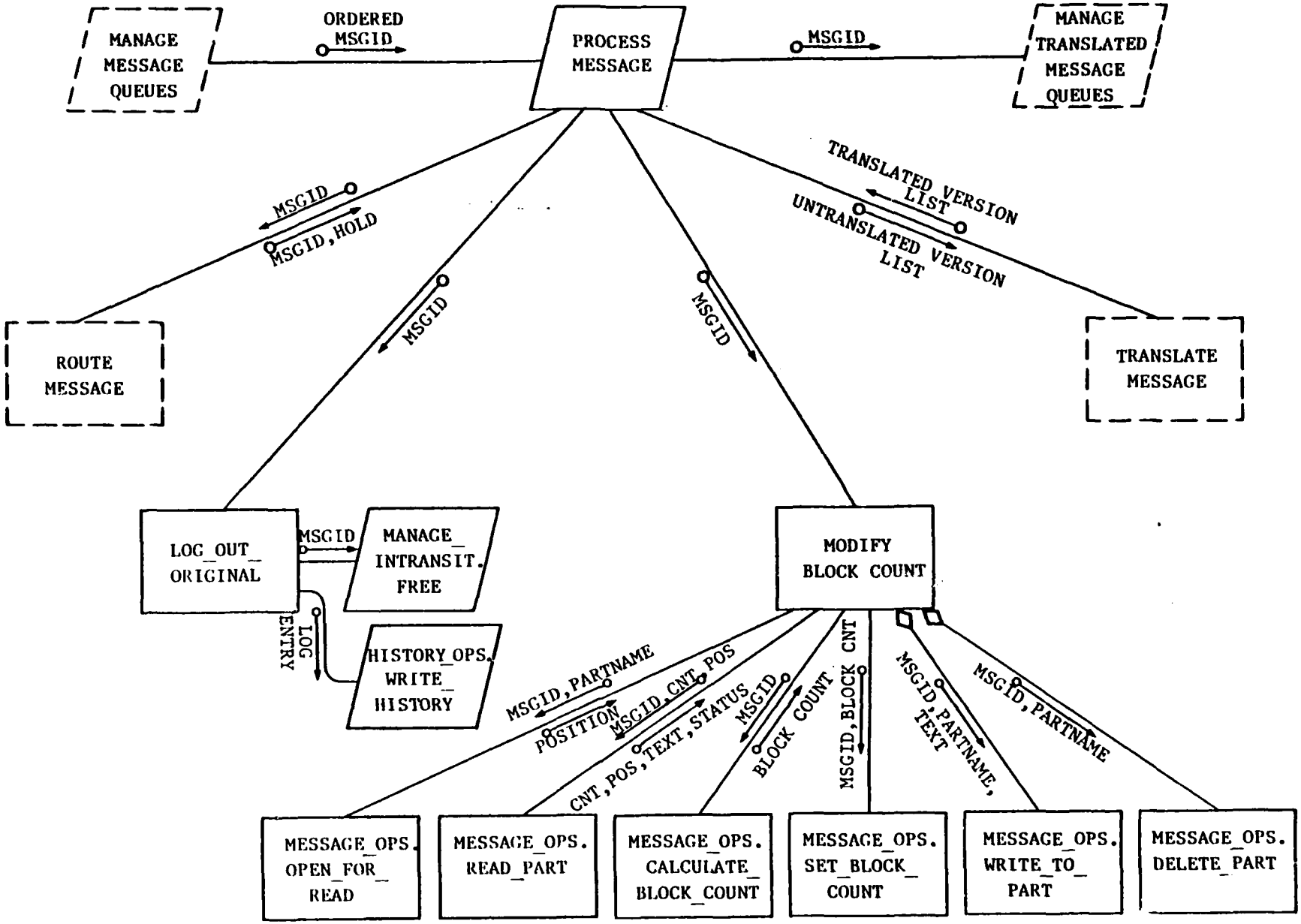



Figure 2.4-24 Process Message

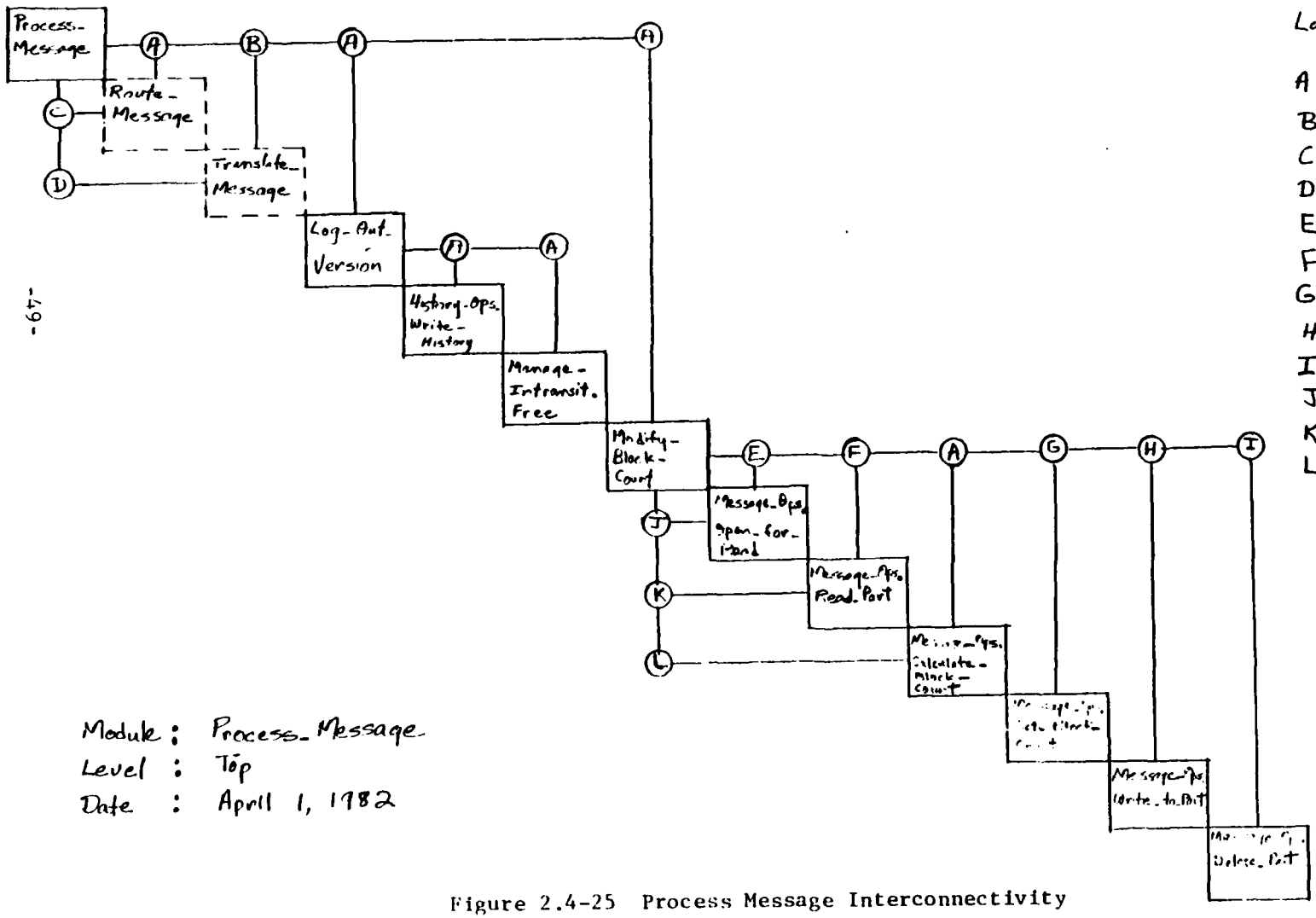



Figure 2.4-25 Process Message Interconnectivity

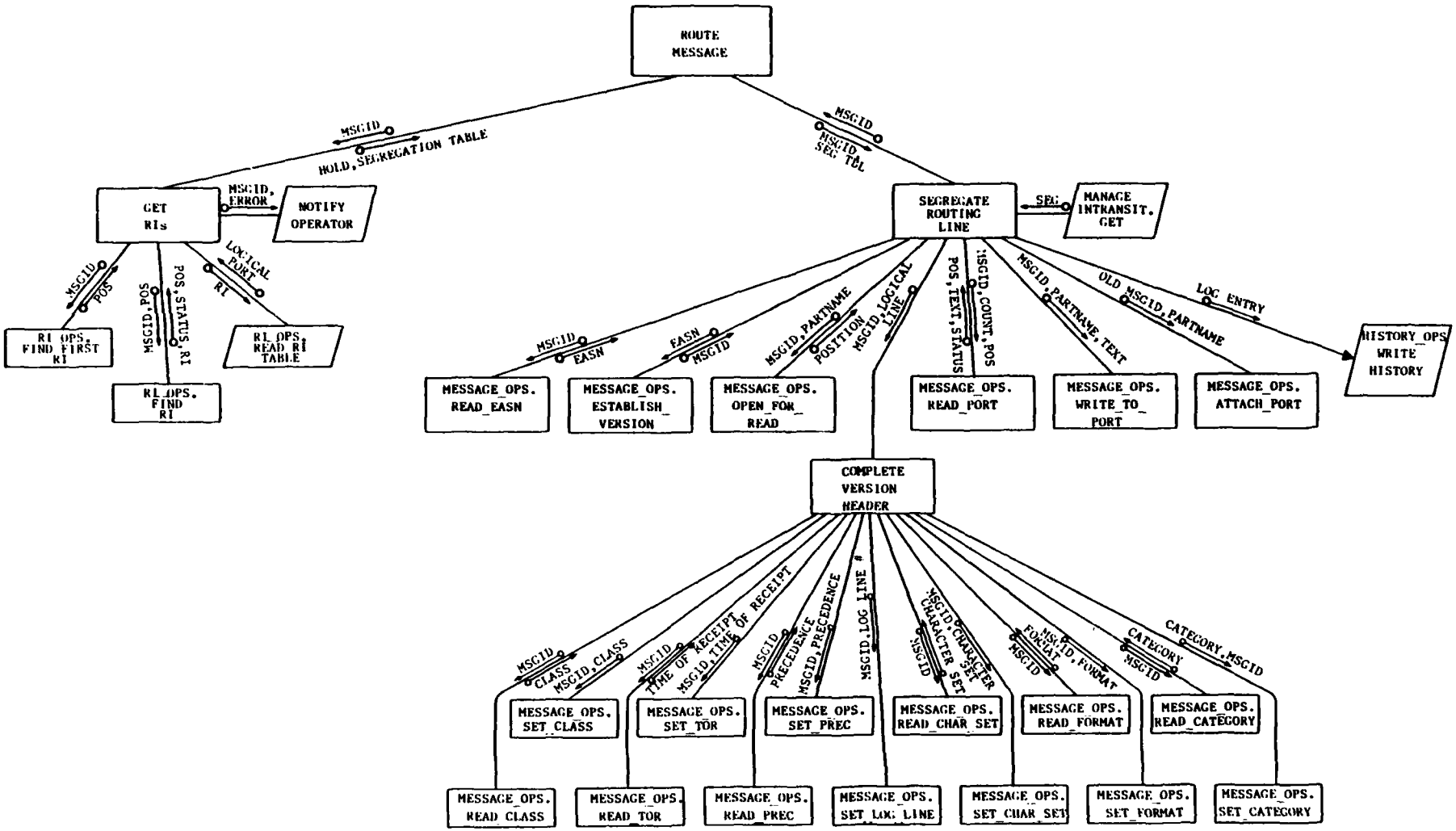
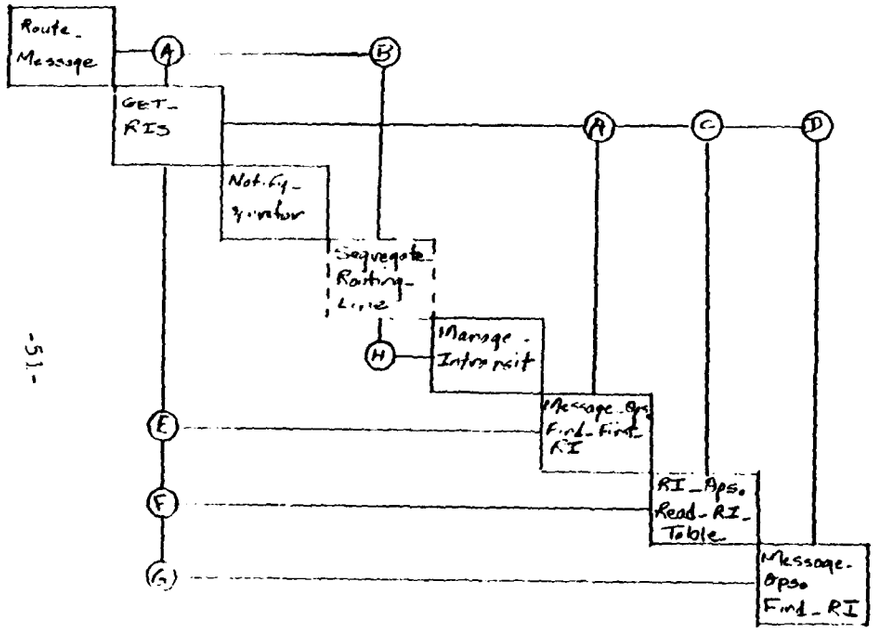



Figure 2.4-26 Route Message

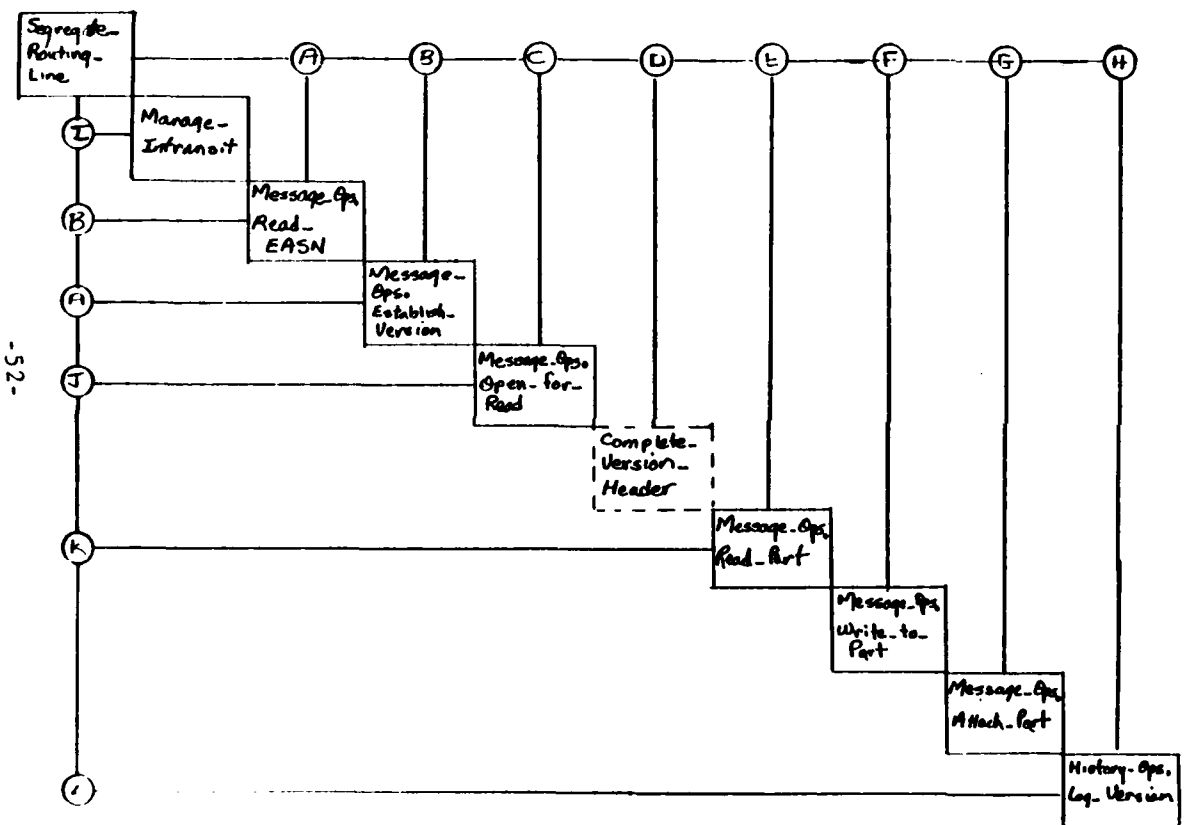



| Label | Data Item(s) |
|---|---|
| A | Message ID |
| B | Message ID, Segment table |
| C | RI |
| D | Message ID, position |
| E | position |
| F | logical port |
| G | position, status, RI |
| H | Segment |

Date : April, 1, 1982



Figure 2.4-27 Route Message Interconnectivity




| Label | Data Item(s) |
|---|---|
| A | Message ID |
| B | EASN |
| C | Message ID, partname |
| D | Message ID, Logical line |
| E | Message ID, count, position |
| F | Message ID, partname, text |
| G | Old Message ID, partname |
| H | Version Message ID |
| I | Segment |
| J | position |
| K | position, text, status |
| L | status |

Module : Segregate_ Routing_ Line
Date : April 2, 1982

Figure 2.4-28 Segregate Routing Line Interconnectivity

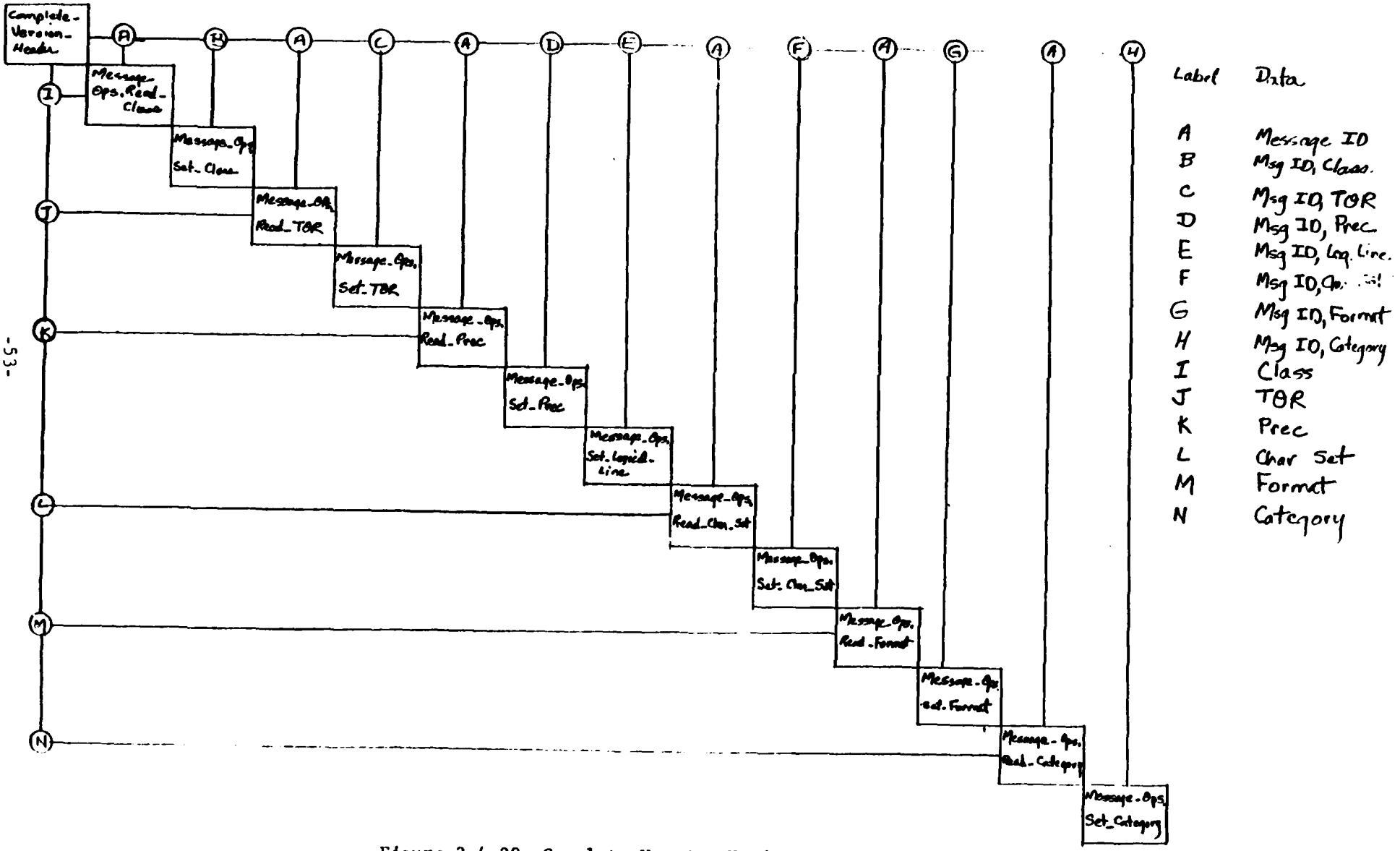



Figure 2.4-29 Complete Version Header Interconnectivity

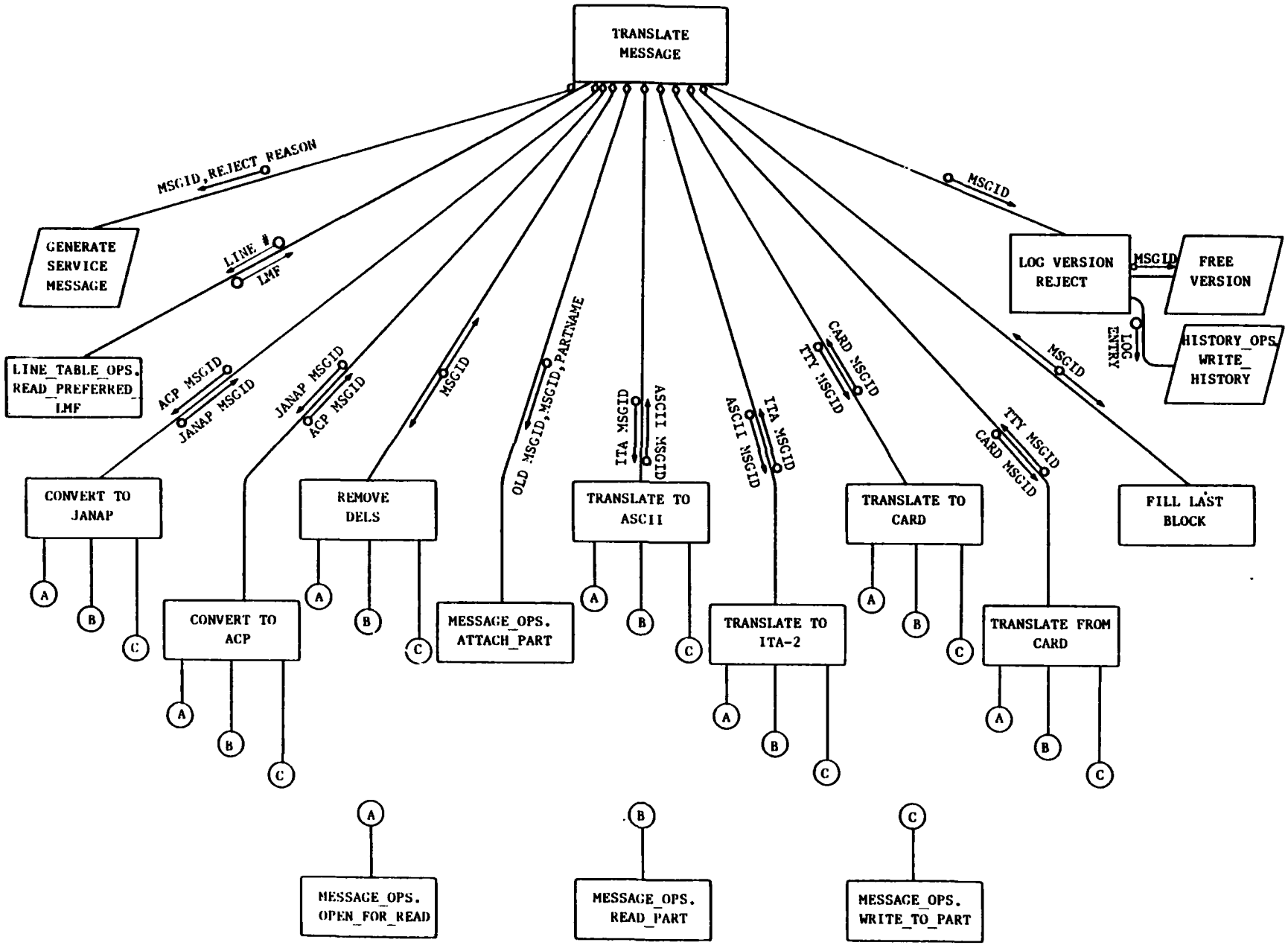
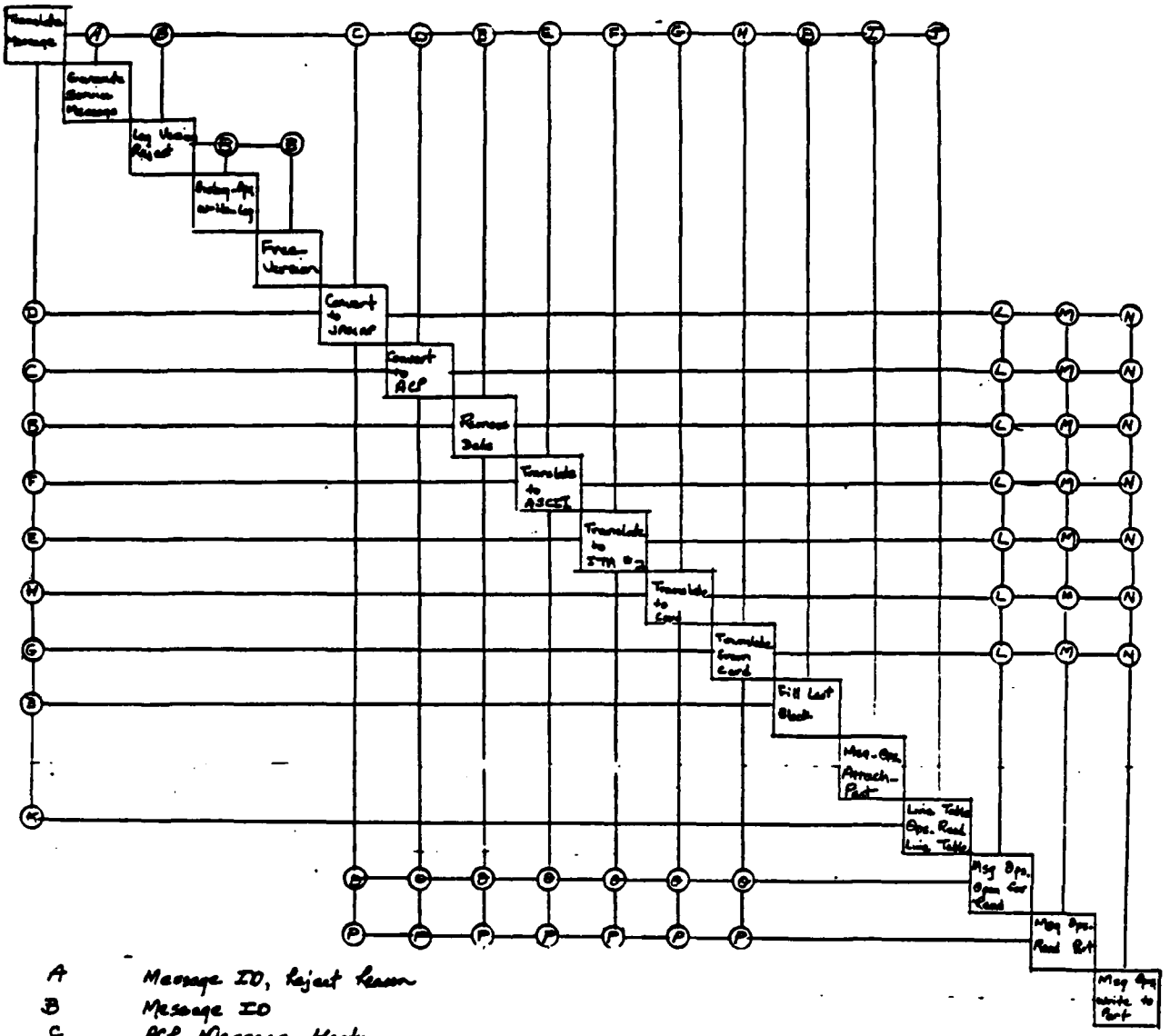



Figure 2.4-30 Translate Message



| | |
|---|---|
| A | Message ID, Reject Reason |
| B | Message ID |
| C | ACP Message Header |
| D | JANAP Message ID |
| E | ITA Message ID |
| F | ASCII Message ID |
| G | TTY Message ID |
| H | Card Message ID |
| I | Bll Message ID, Message ID, portname |
| J | Line |
| K | LMF |
| L | Message ID, portname |
| M | Message ID, count, position |
| N | Message ID, portname, text |
| O | Position |
| P | Count, position, text, status |

Module : Translate Message

Date : April 5, 1952

Figure 2.4-31 Translate Message Interconnectivity

### 2.4.4 Output Message

The Output Message function is illustrated in Figures 2.4-32 through 2.4-39.

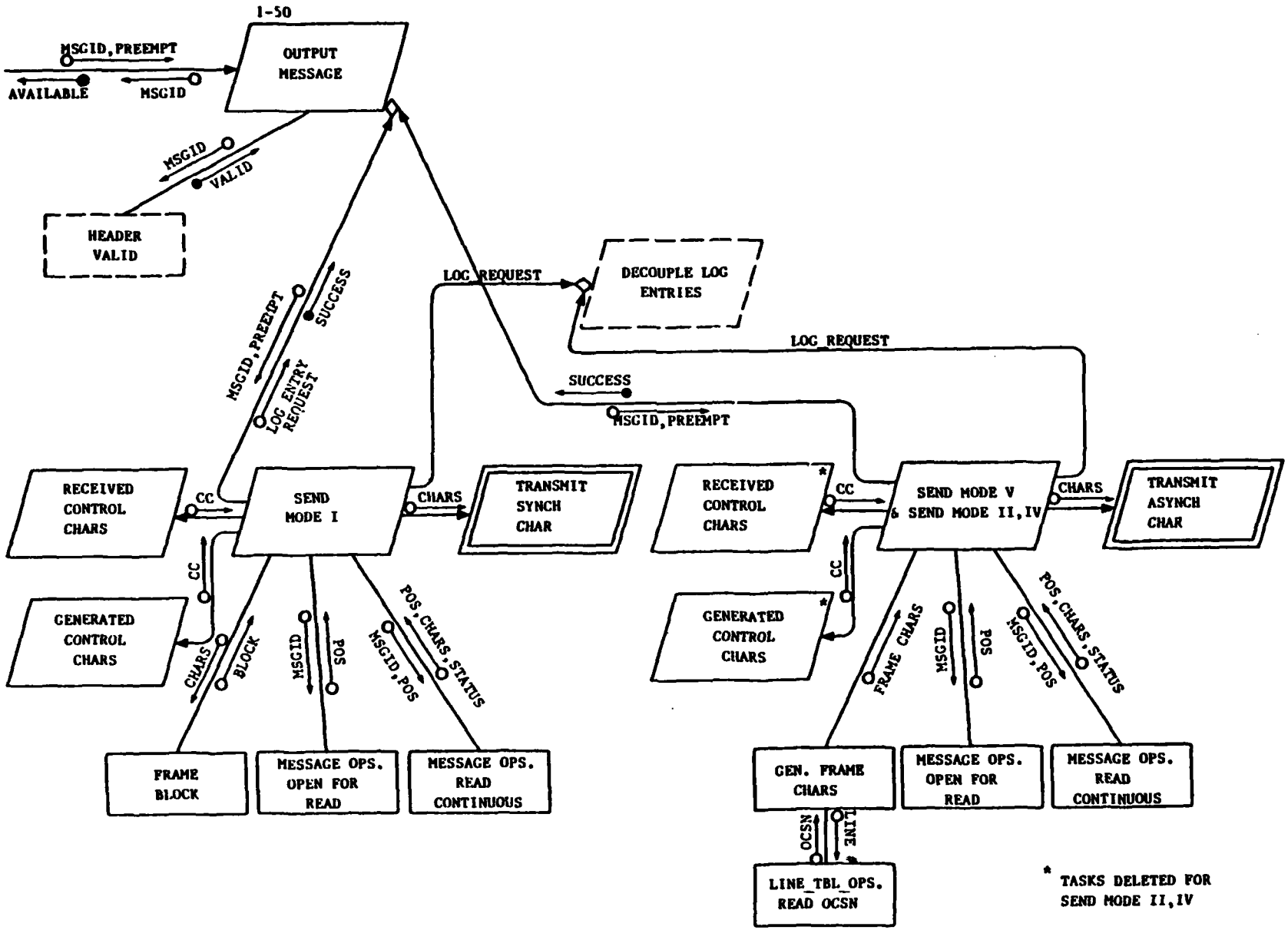
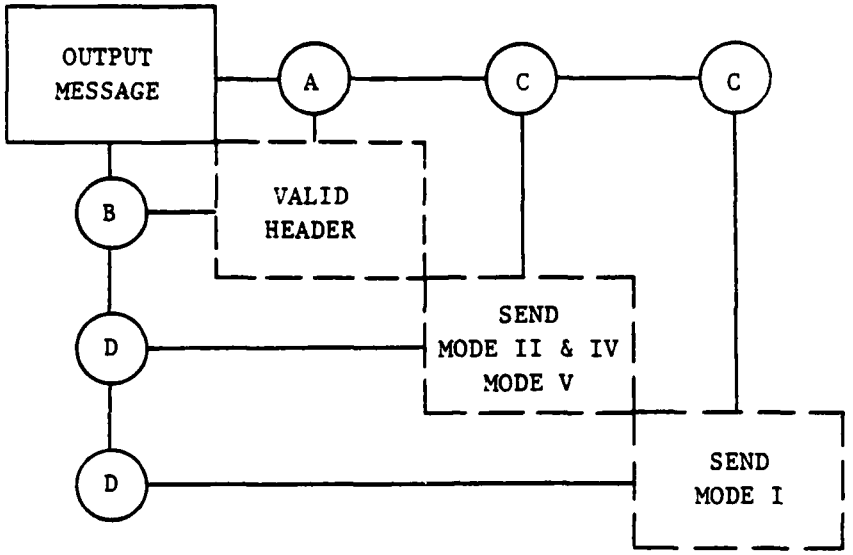
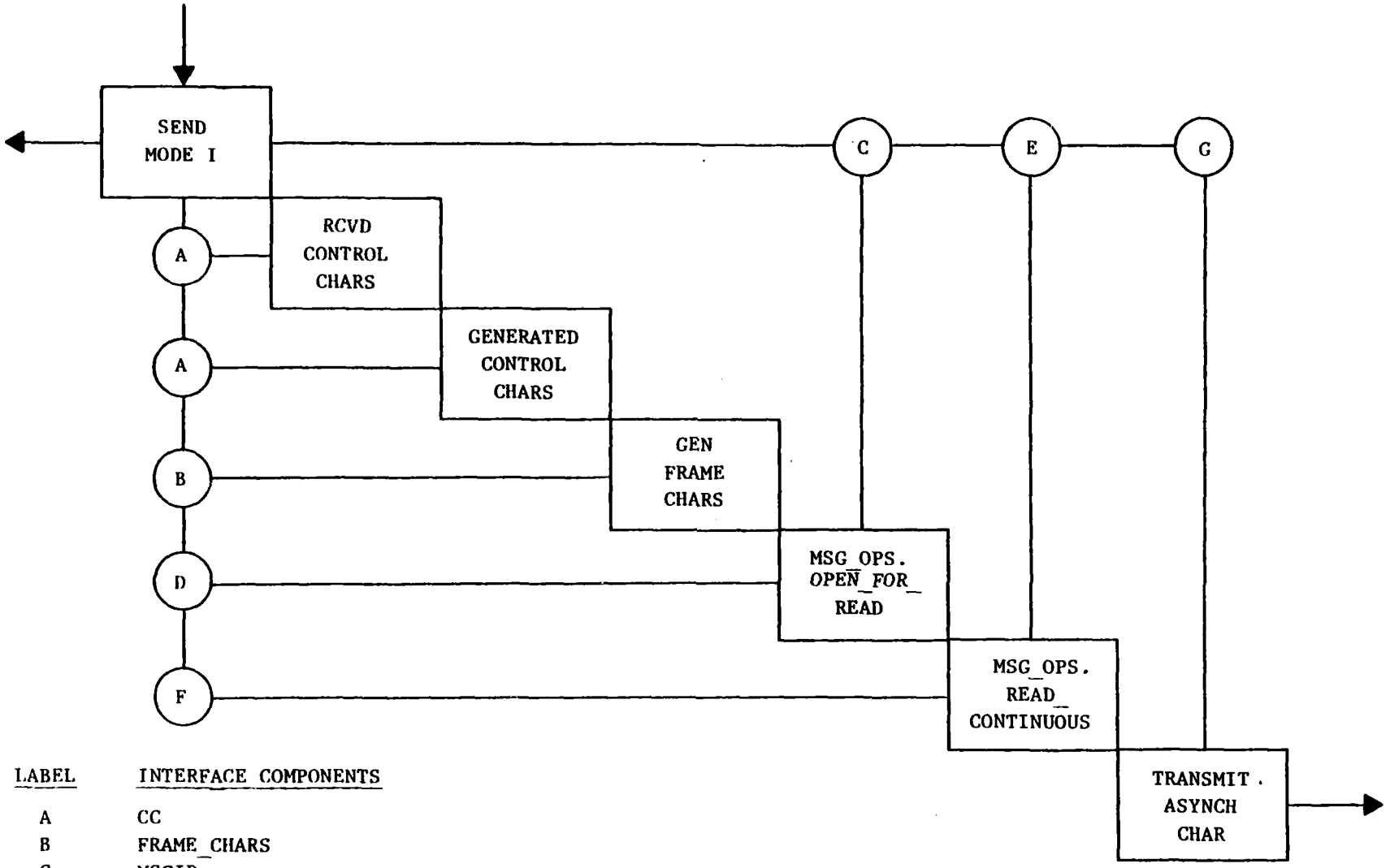



Figure 2.4-32 Output Message



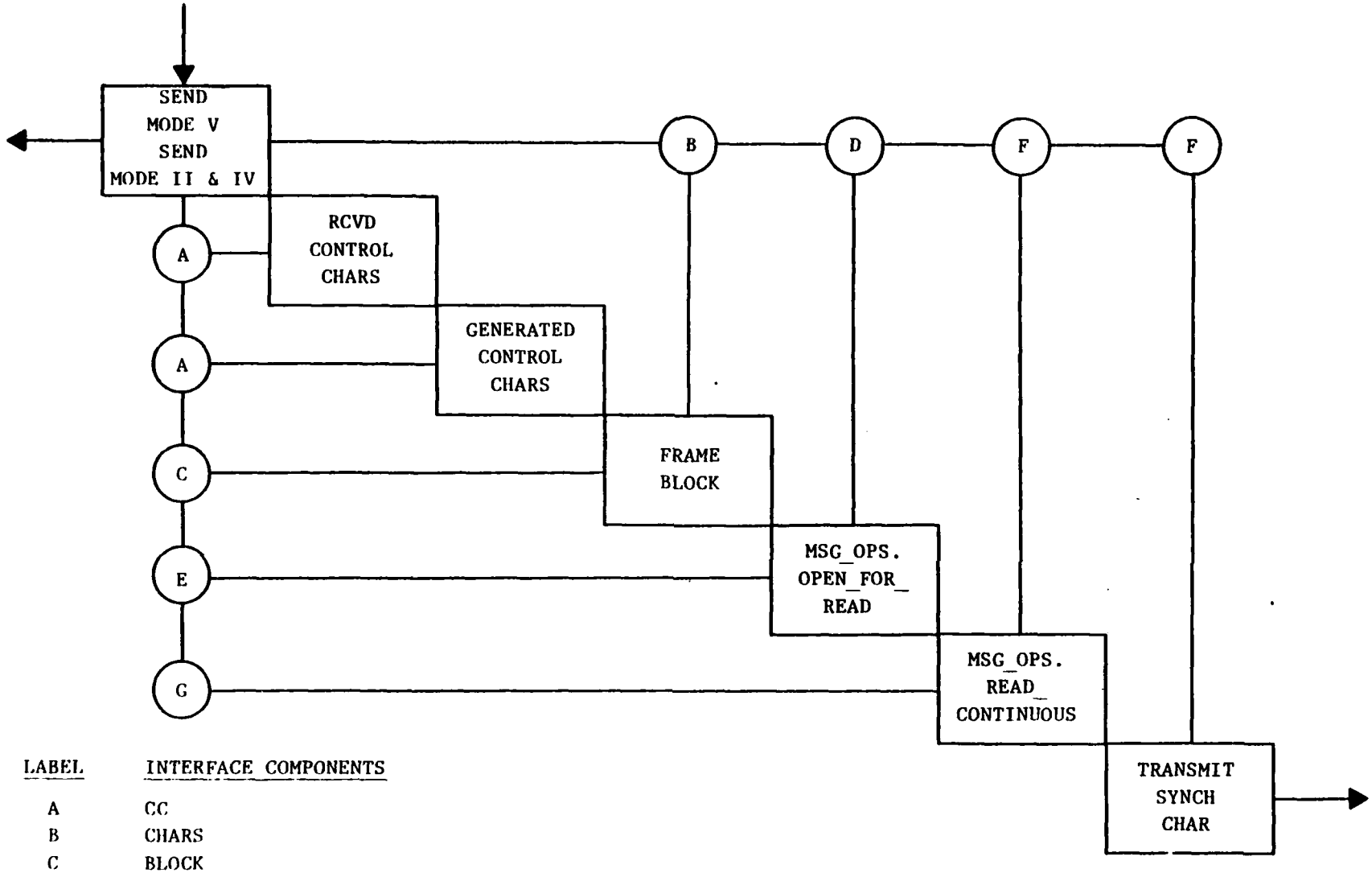

| LABEL | INTERFACE COMPONENTS |
|---|---|
| A | MSGID |
| B | VALID_FLAG |
| C | MSGID,PREEMPT |
| D | LOG_ENTRY_REQUEST,SUCCESS |

Figure 2.4-33 Output Message Interconnectivity




| LABEL | INTERFACE COMPONENTS |
|---|---|
| A | CC |
| B | FRAME_CHARS |
| C | MSGID |
| D | POSITION |
| E | MSGID,POSITION |
| F | POSITION,CHARS,STATUS |
| G | CHARS |

Figure 2.4-34 Send Mode I Interconnectivity




| LABEL | INTERFACE COMPONENTS |
|---|---|
| A | CC |
| B | CHARS |
| C | BLOCK |
| D | MSGID |
| E | POSITION |
| F | MSGID,POSITION |
| G | POSITION,CHARS,STATUS |

Figure 2.4-35 Send Mode V and Send Mode II & IV Interconnectivity

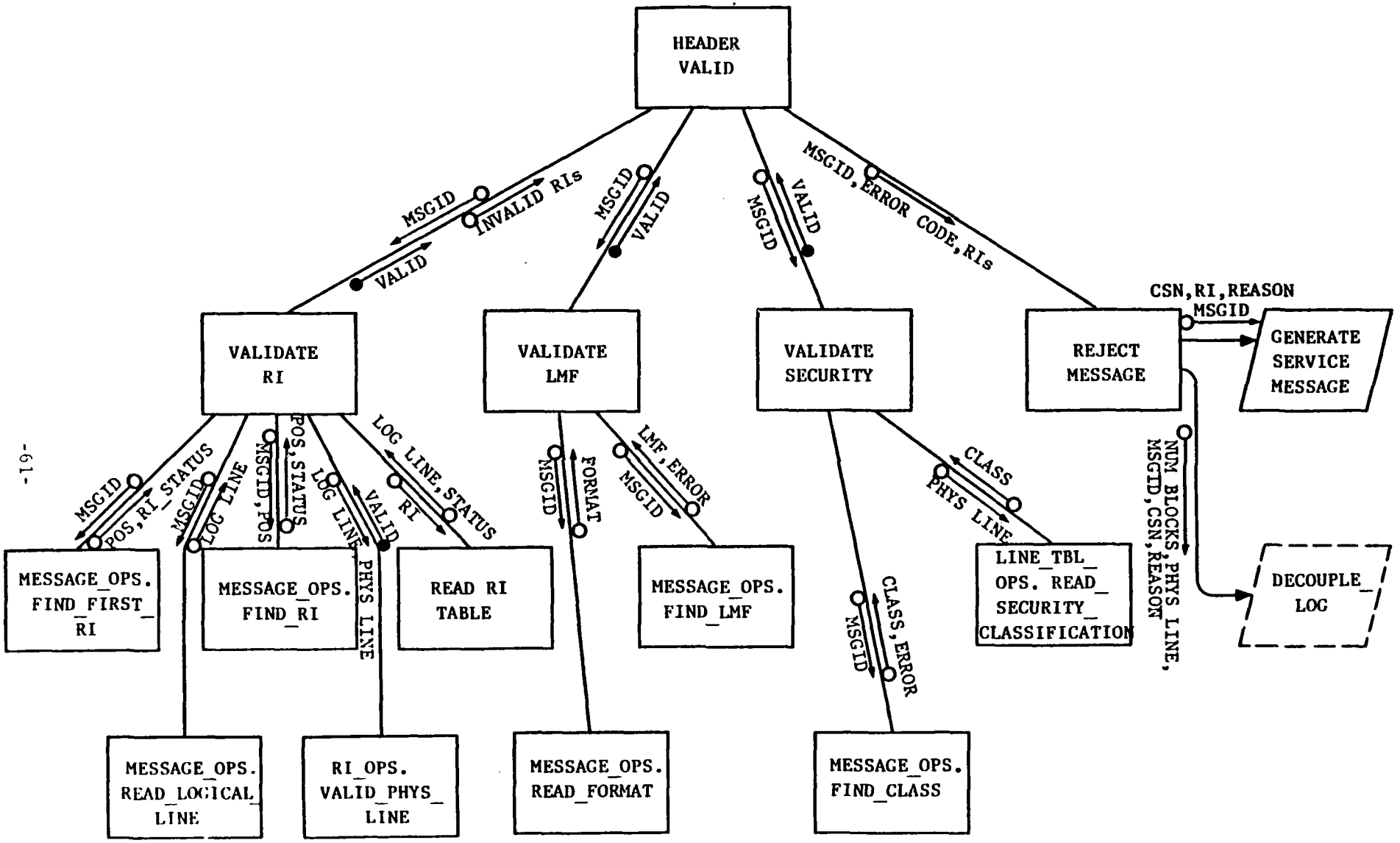
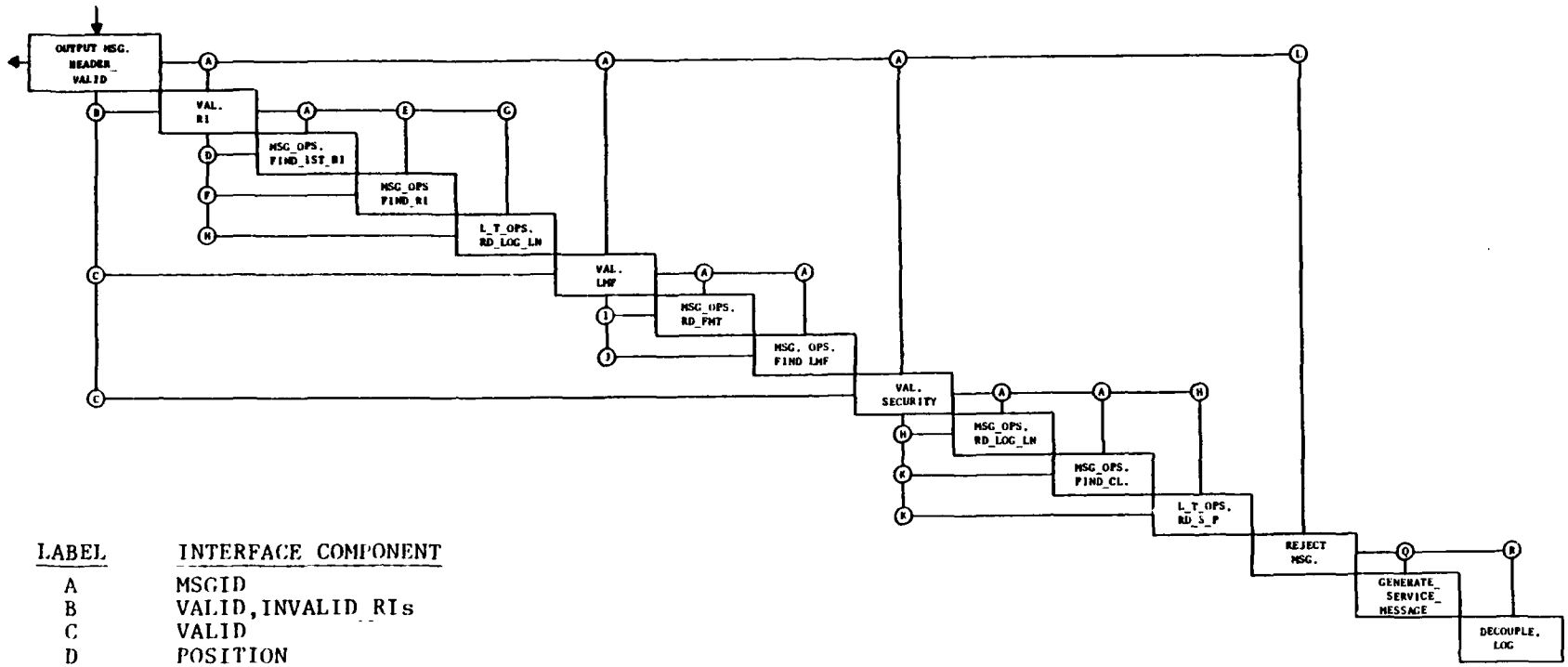



Figure 2.4-36 Header Valid




| LABEL | INTERFACE COMPONENT |
|---|---|
| A | MSGID |
| B | VALID,INVALID_RIs |
| C | VALID |
| D | POSITION |
| E | MSGID,POSITION |
| F | POSITION,RI,STATUS |
| G | RI |
| H | LOGICAL_LINE |
| I | FORMAT |
| J | LMF |
| K | CLASSIFICATION |
| L | MSGID,ERROR_CODE,RIs |
| M | DATE_TIME |
| N | RI,REASON,MSGID |
| O | LOG_ENTRY |
| P | MSGID,ERROR_CODE |
| Q | CSN,RI,REASON,MSGID |
| R | #_BLOCKS,PHYS_LINE_#,MSGID,CSN,REASON |

Figure 2.4-37 Header Valid Interconnectivity

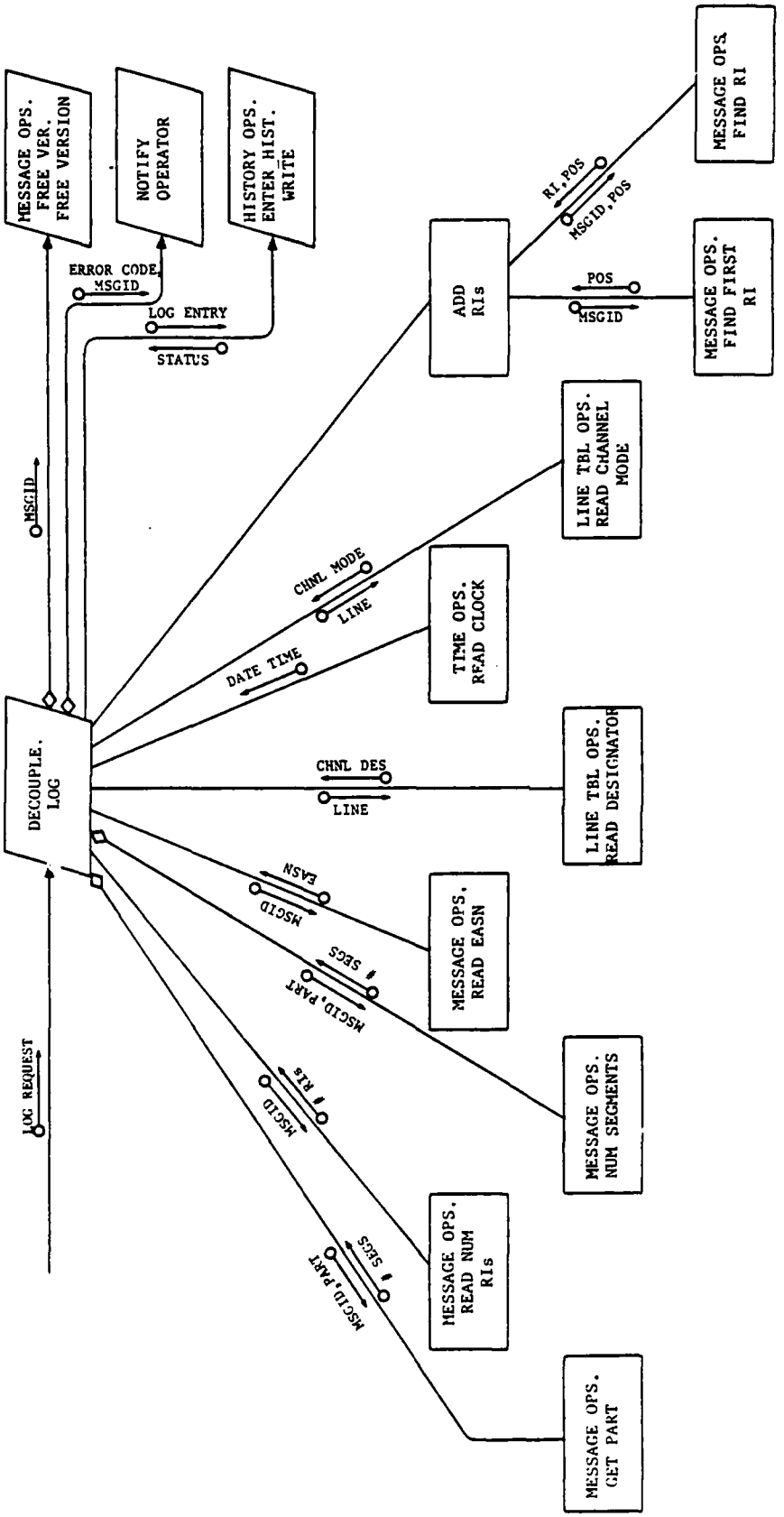
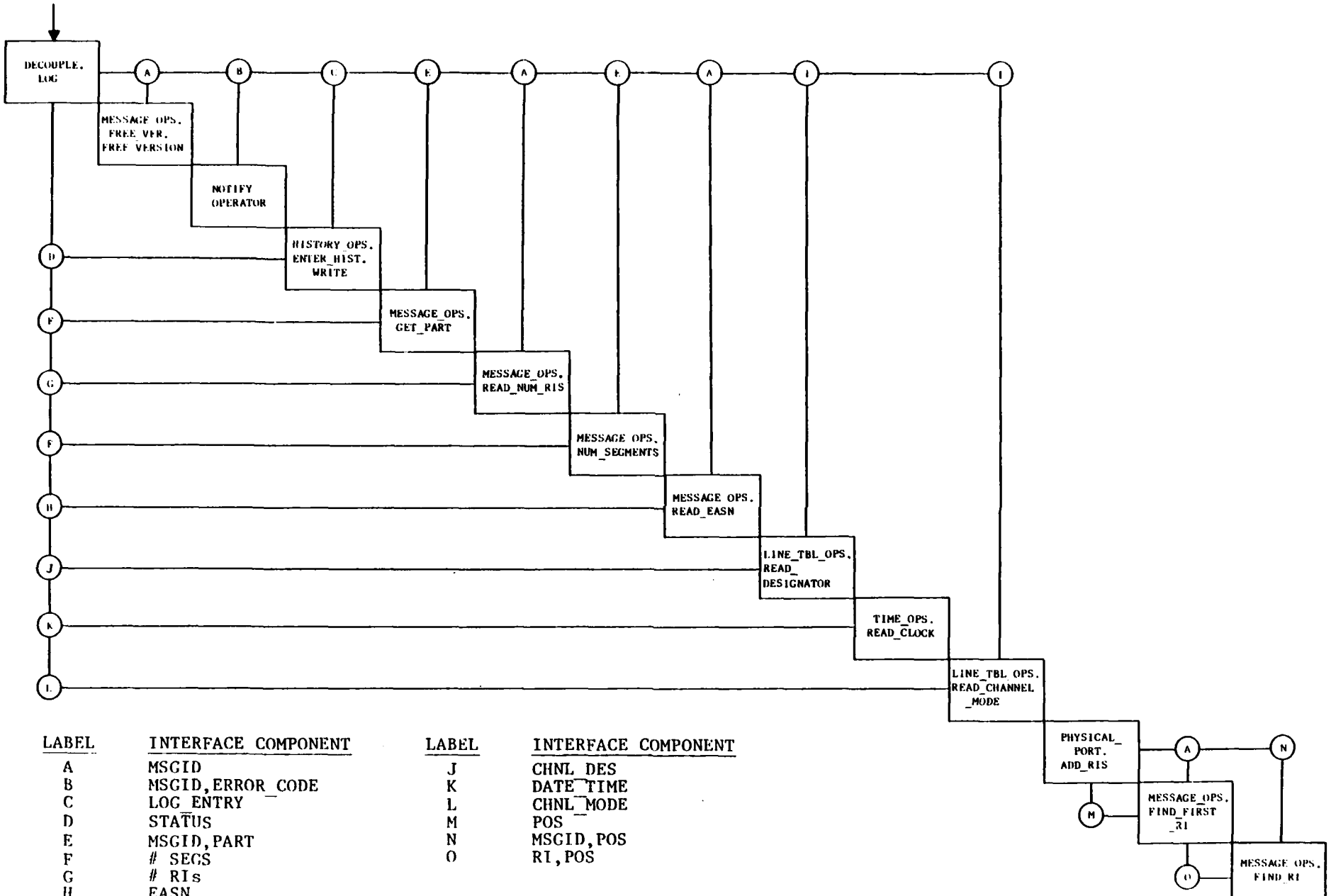



Figure 2.4-38 Decouple Log




| LABEL | INTERFACE COMPONENT | LABEL | INTERFACE COMPONENT |
|---|---|---|---|
| A | MSGID | J | CHNL_DES |
| B | MSGID,ERROR_CODE | K | DATE_TIME |
| C | LOG_ENTRY | L | CHNL_MODE |
| D | STATUS | M | POS |
| E | MSGID,PART | N | MSGID,POS |
| F | # SEGS | O | RI,POS |
| G | # RIs | | |
| H | EASN | | |
| I | LINE | | |

Figure 2.4-39 Decouple Log Interconnectivity

### 2.4.5 Manage Intransit, Manage Overflow

The Manage Intransit, Manage Overflow function is illustrated in Figures 2.4-40 through 2.4-41.

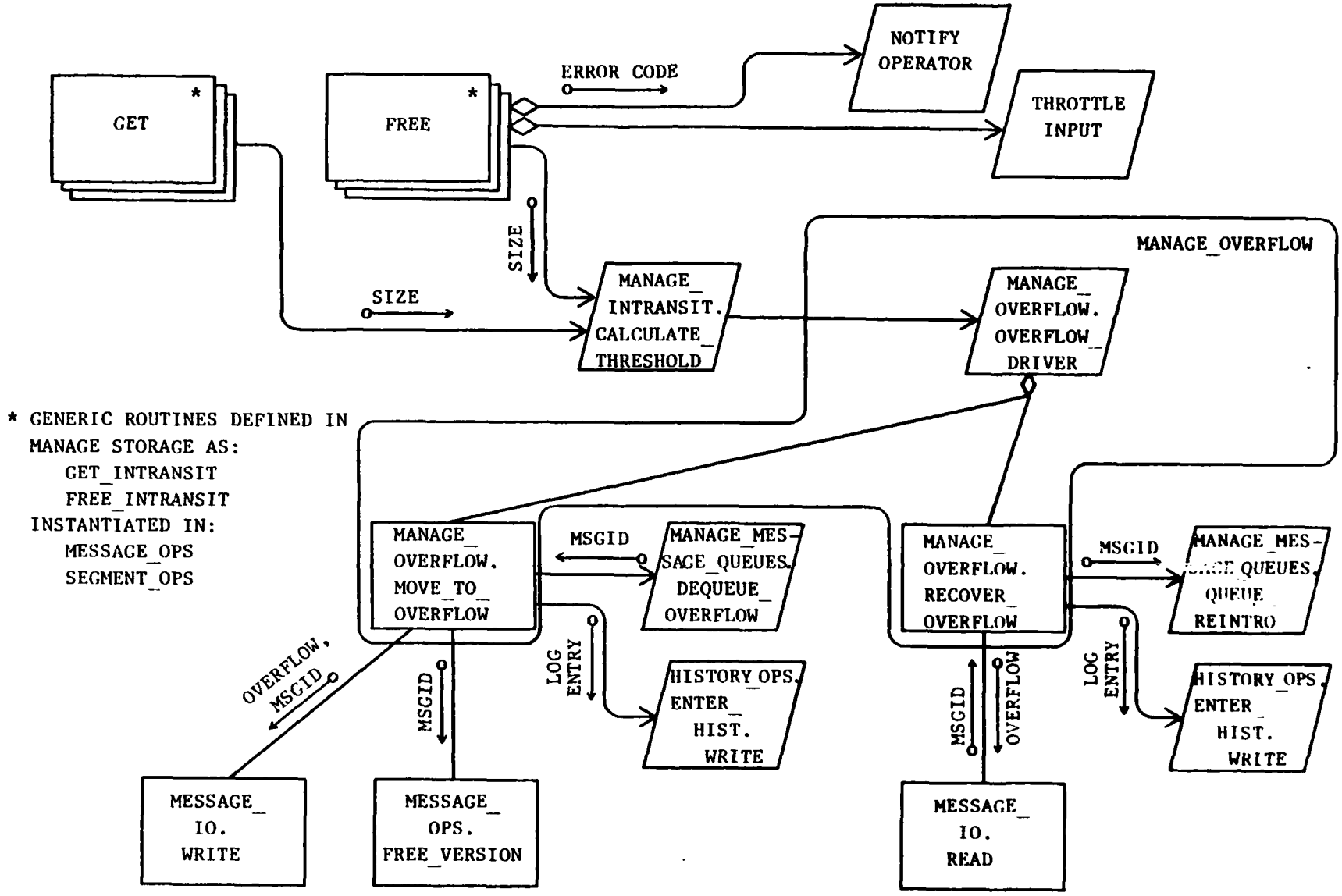



Figure 2.4-40 MESSAGE STORAGE ALLOCATION/DEALLOCATION

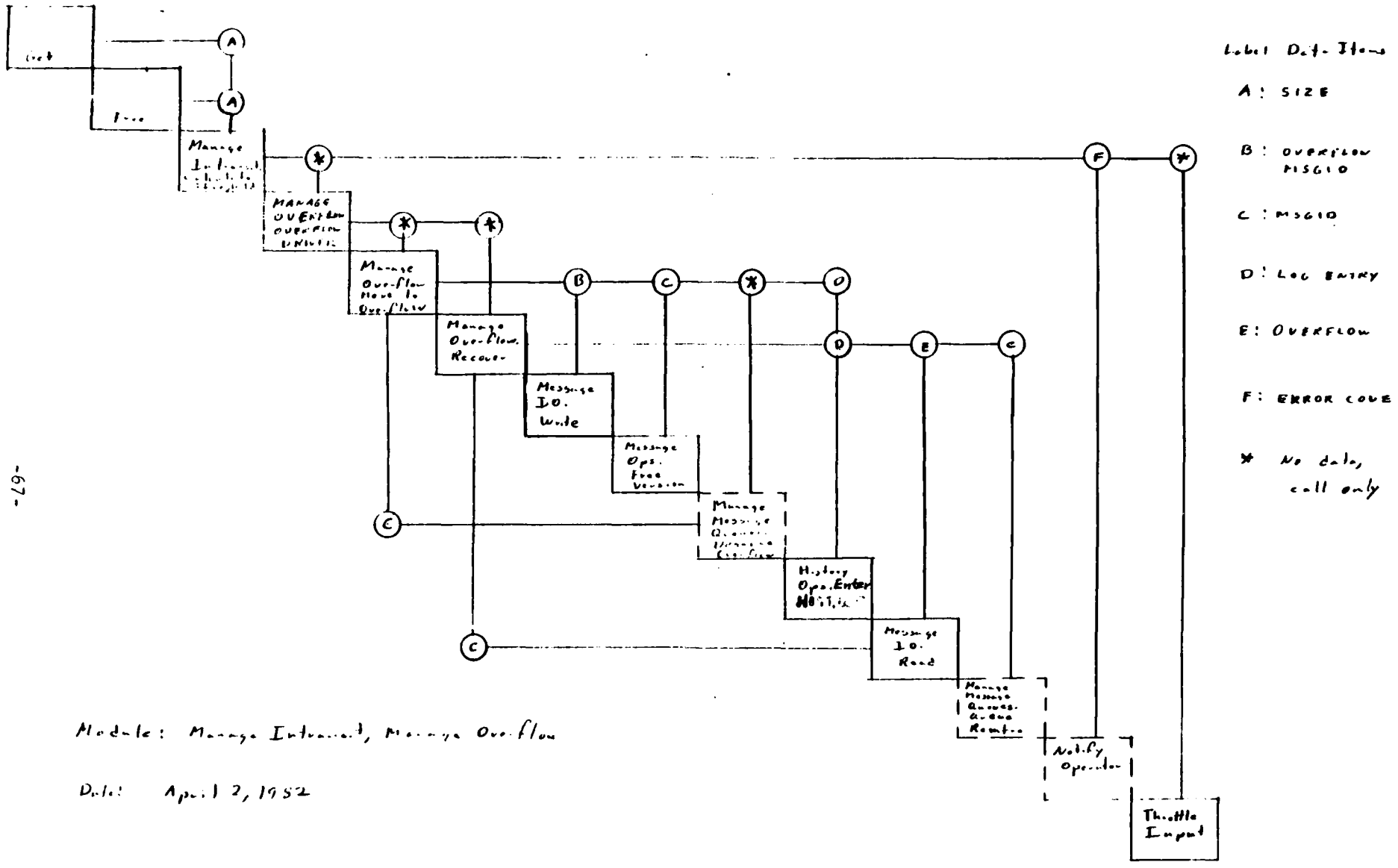



Figure 2.4-41 Manage Intransit, Manage Overflow Interconnectivity

## 2.5 Data Structure Diagrams

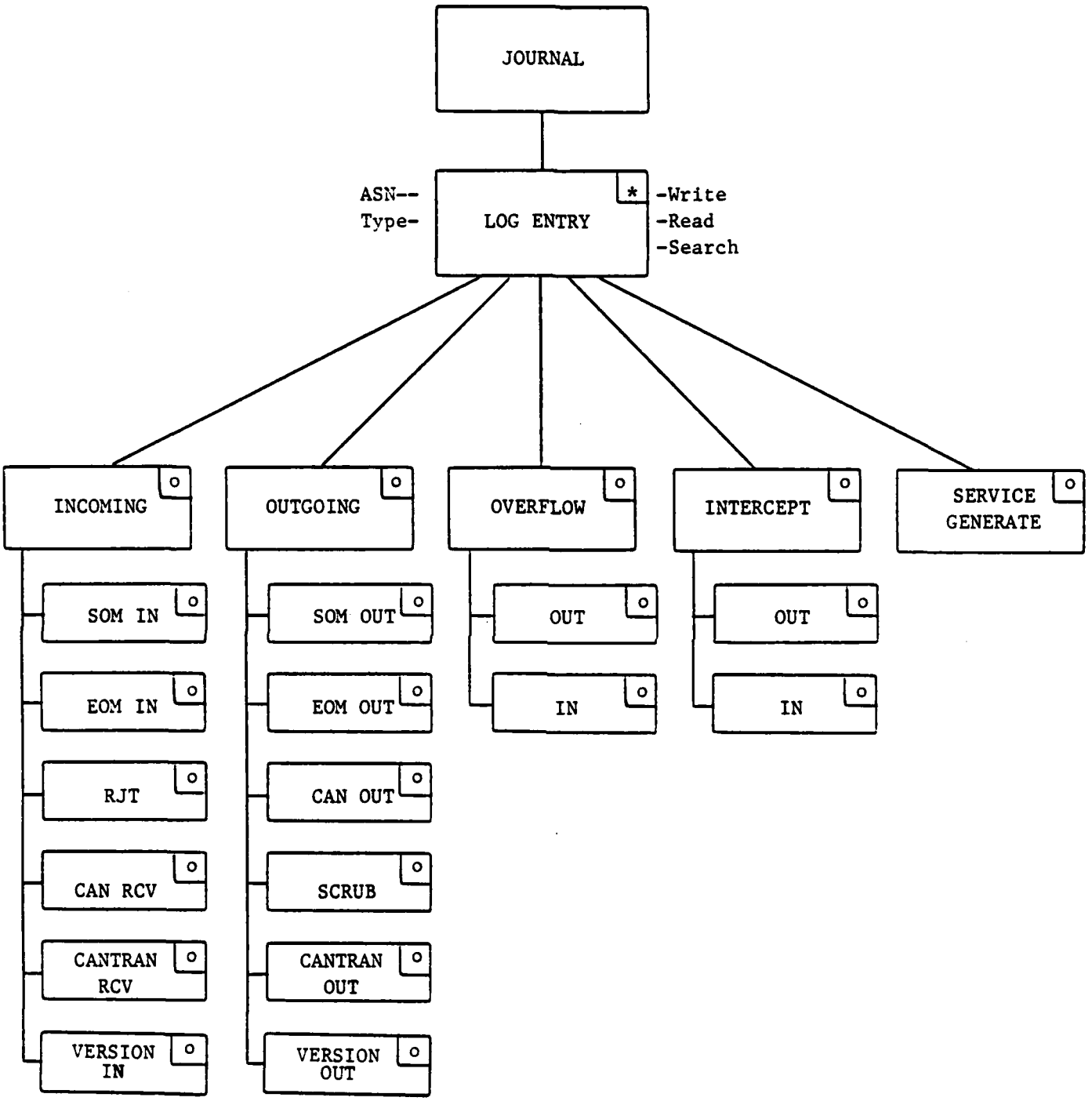



Figure 2.5-1 Journal Object

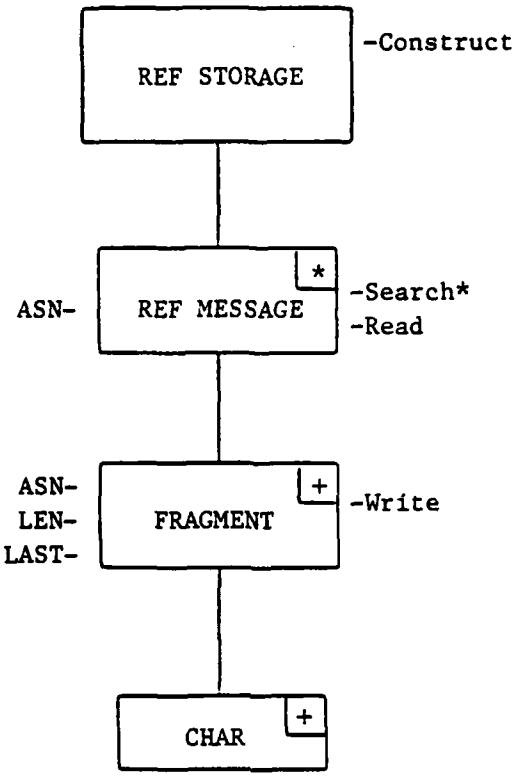



Figure 2.5-2 Reference Storage Object

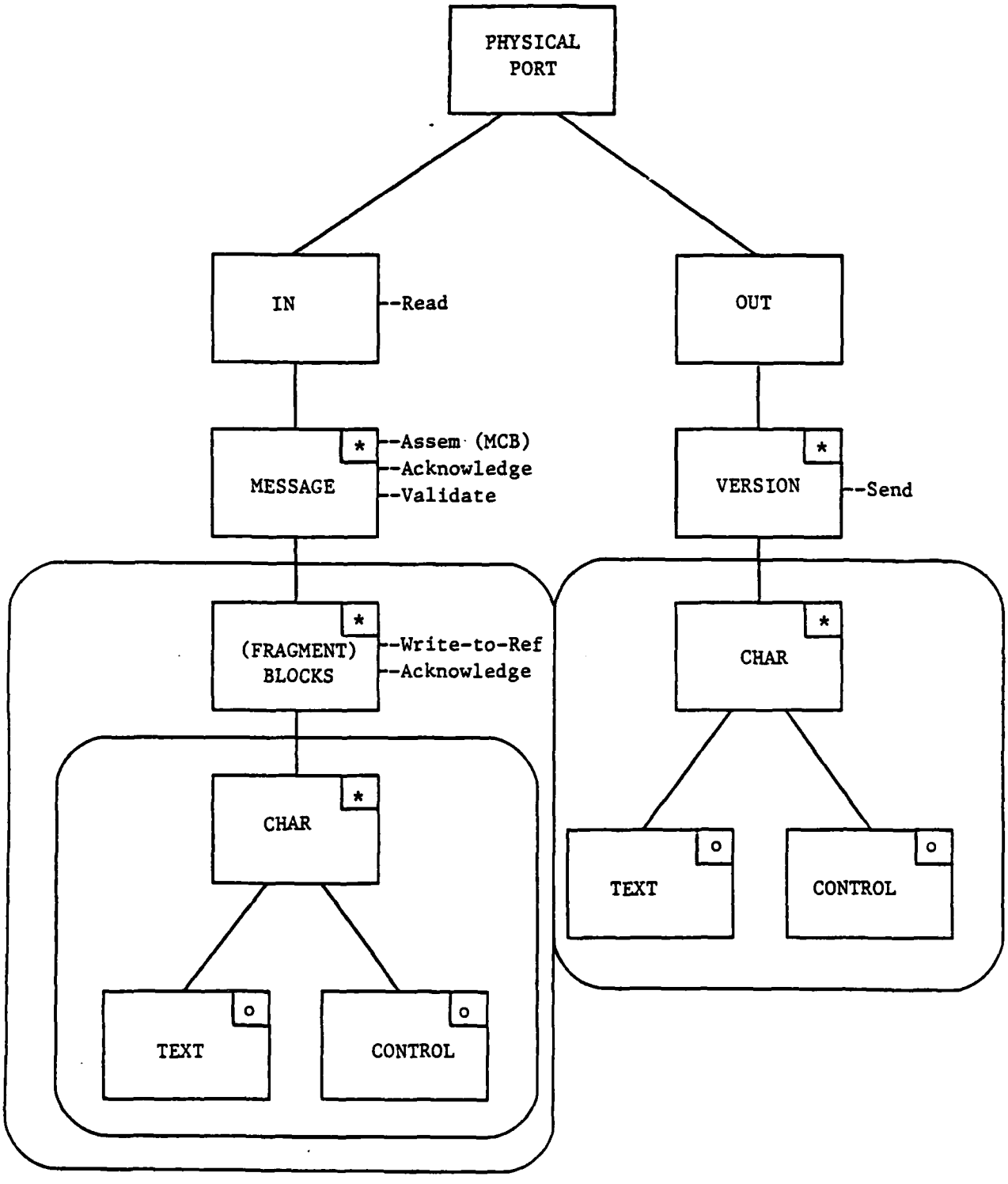



Figure 2.5-3 Physical Port Object

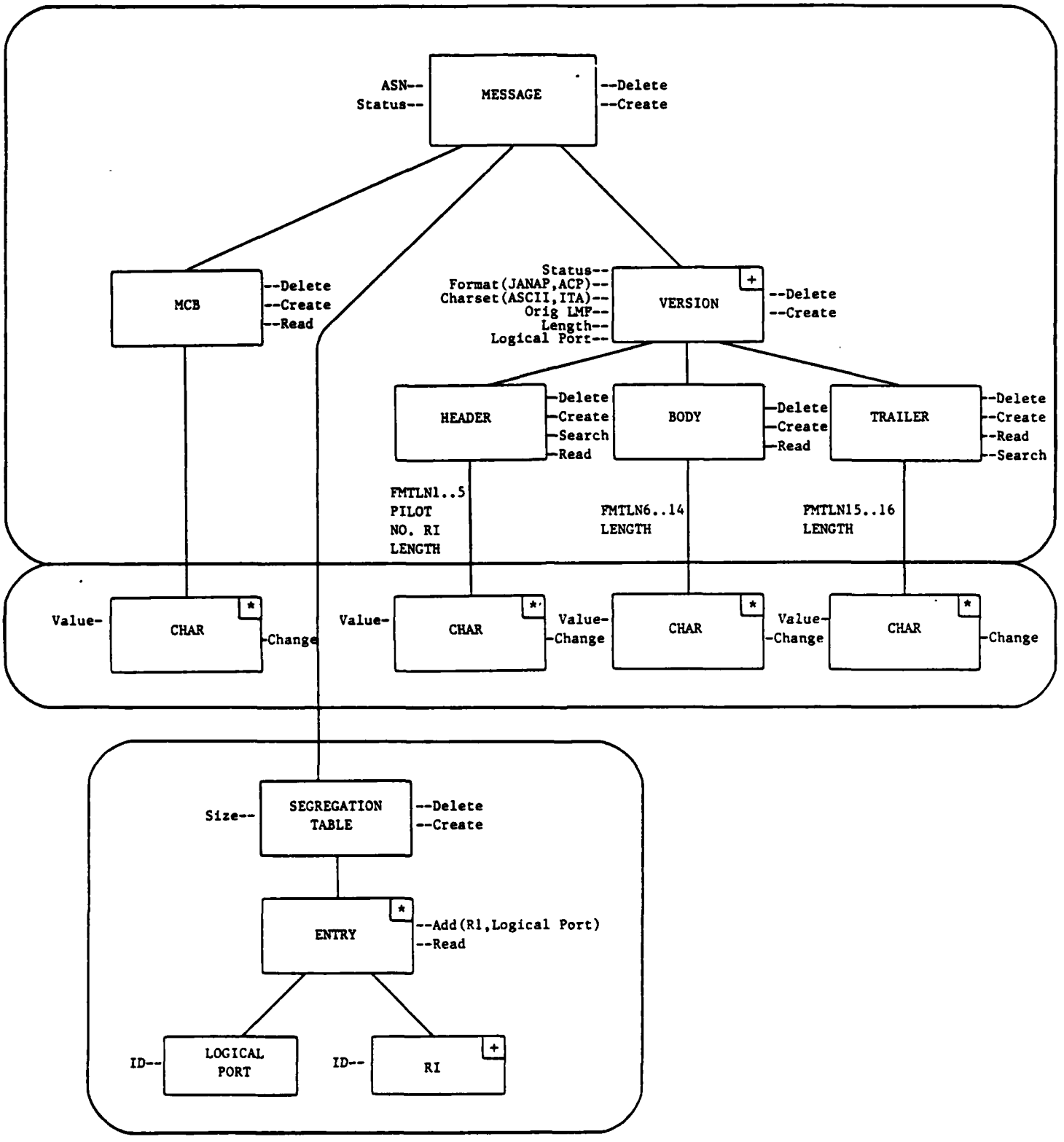



Figure 2.5-4 Message Object

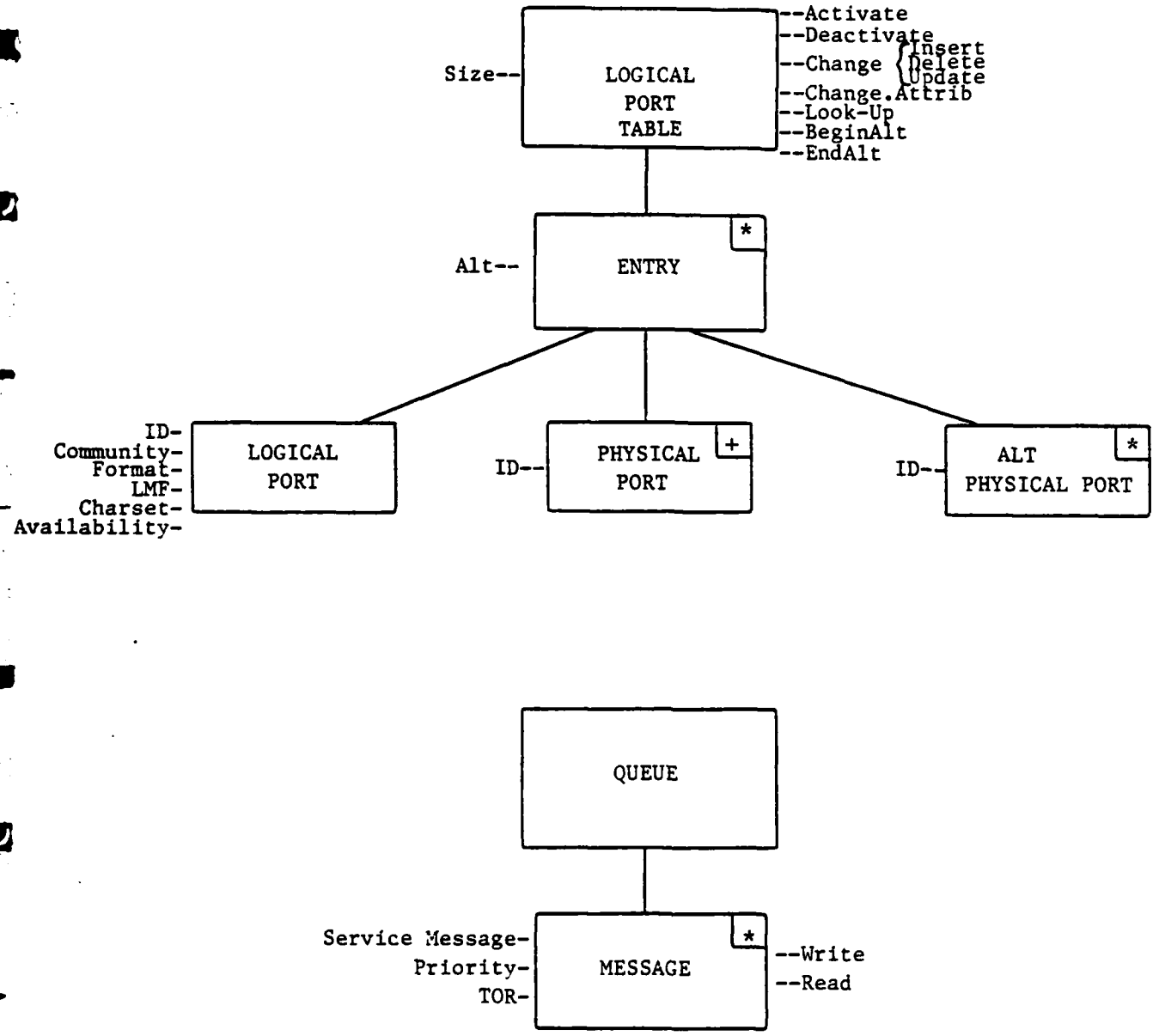



Figure 2.5-5 Logical Port Table and Queue Objects

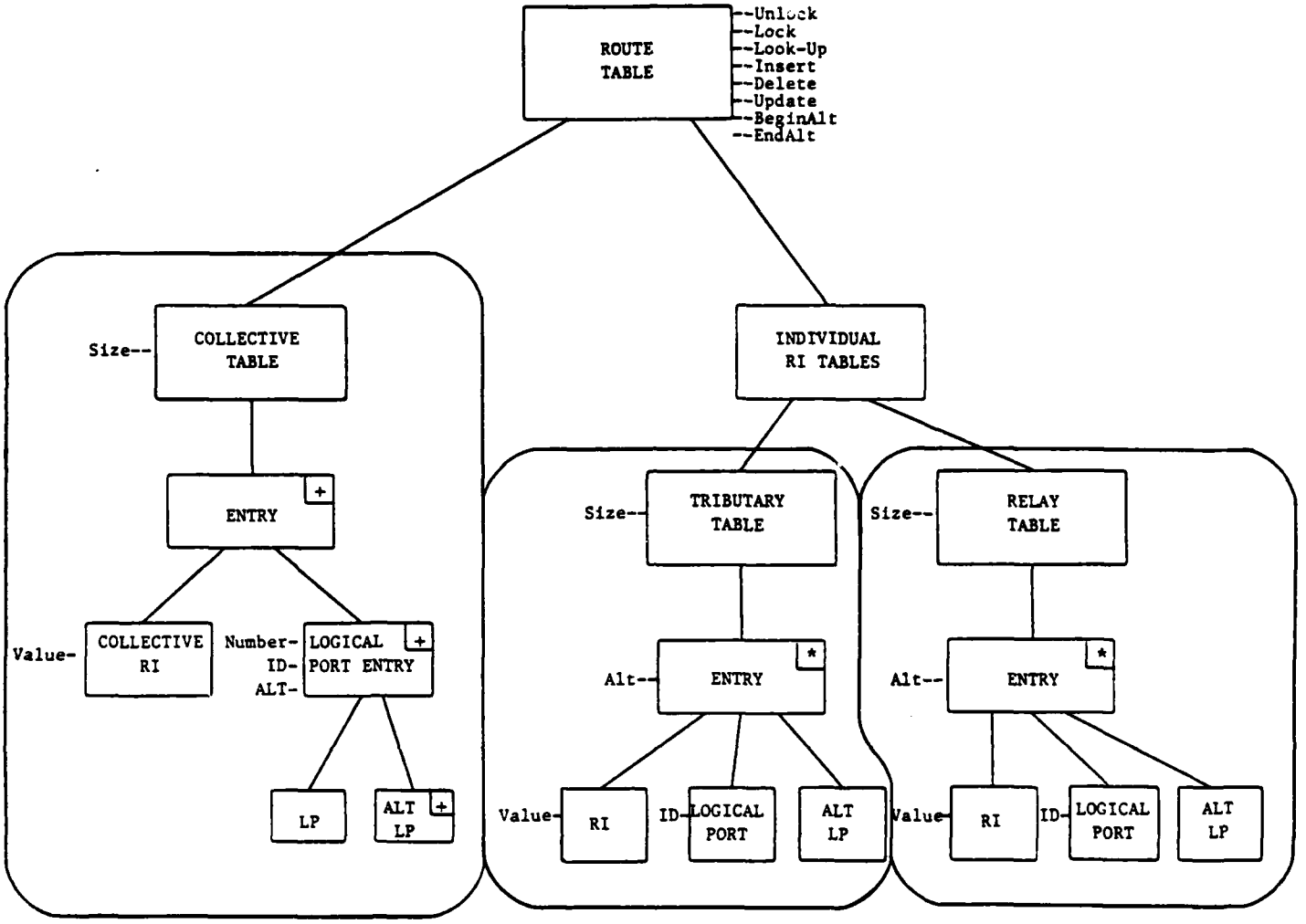



Figure 2.5-6 Route Table Object

```
HIGH LEVEL PORT TASK

    LOOP
      ASSEMBLE.MESSAGE                  -- CONSTRUCT NEXT MESSAGE
      WRITE.INCOMINGQ (MESSAGE)         -- PLACE IN Q
    ENDLOOP
```

Figure 2.5-7 Example of Message Input Utilizing Object Oriented Structure

```
PROCESS MESSAGE

    LOOP                                    -- LOOP FOREVER
      READ.INCOMINGQ                        -- GET MESSAGE
      LOOK_UP.ROUTE_TABLE (MESSAGE)         -- DETERMINE ROUTING LIST
      FOR EACH ROUTE_INDICATOR              -- FOR EACH DESTINATION
        CREATE.MESSAGE.VERSION (LMF)        -- CREATE OUTPUT FORMAT
        WRITE.OUTGOINGQ (MESSAGE.VERSION)   -- SEND THE MESSAGE
      ENDFOR
    ENDLOOP                                 -- END LOOP
```


Figure 2.5-8 Example of Process Message Utilizing Object Oriented Structure

# 3. Detailed Design

## 3.1 Ada Unit Specifications

### 3.1.1 Message Input

```
with SEGMENT_OPS, LINE_TBL_OPS, MESSAGE_OPS; use SEGMENT_OPS,
    MESSAGE_OPS;

package INPUT_PORT is

  task type RECEIVE_VALID_MESSAGE is
    entry INITIATE(PHYSICAL_PORT:LINE_TBL_OPS.PHYSICAL_LINE);
    entry TERM(PHYSICAL_PORT:LINE_TBL_OPS.PHYSICAL_LINE);
                                                          end RECEIVE

  type SEG_STATUS is (GOOD,CAN,CANTRAN,EXPIRED_PAUSE);
  type RCV_STATUS is (VALID,HOLD,STORE_FAILED,BAD_MCB,BAD_RIS,
      BAD_MODE, BAD_SECURITY);

  task type BUILD_MESSAGE_FROM_SEGMENTS is
    entry BUILD_SEG(NEW_SEG:SEG_PTR;SEG_RESULT:SEG_STATUS);
    entry CHECK_MCB(MCB_SEG:out SEG_PTR);
    entry BUILD_MCB(HDR_SEG:out SEG_PTR);
    entry COMPLETE_MCB(MCB_SEG:SEG_PTR;TRL_SEG:out SEG_PTR);
    entry VALID_MCB(MCB_RESULT:RCV_STATUS);
    entry STORE_REF(STORE_SEG:out SEG_PTR);
    entry STORE_FAIL(FAIL_SEG: SEG_PTR);
    entry CHECK_SEG(VALID_SEG:out SEG_PTR);
    entry VALID_SEG(SEG_RESULT:RCV_STATUS);
    entry POST_STATUS(MSG_STATUS:out RCV_STATUS);
    entry RECEIVE_MESSAGE(MESSAGE:out MSGID;MSG_STATUS:out
        RCV_STATUS);
  end BUILD_MESSAGE_FROM_SEGMENTS;

end INPUT_PORT;
```

```
with GLOBAL_TYPES,MANAGE_STORAGE; use GLOBAL_TYPES,MANAGE_STORAGE;

package SEGMENT_OPS is
  ------------------------------------------------------------------
  -- NAME: SEGMENT_OPS
  -- PURPOSE: contains the data definitions and operations for the
  --          manipulation of segments, which are used to hold the
  --          actual text of messages.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type CHAR_COUNT is range 0..81;
  type SEGMENT is private;
  type SEG_PTR is access SEGMENT;
  type SEGMENT_NUMBER is range 0..550;

  procedure RESET_SEG(SEG:SEG_PTR);
  -- SETS TEXT BUFFER IN SEGMENT TO EMPTY

  procedure WRITE_SPECIFIC(SEG : SEG_PTR;
                           WHERE : CHAR_COUNT;
                           TEXT:STRING);
  -- PUTS TEXT INTO SEGMENT BUFFER AT SPECIFIC LOCATION
  -- USED BY REWRITE IN MESSAGE_OPS

  procedure ADD_TEXT(SEG : SEG_PTR; TEXT : STRING);
  -- PUTS TEXT INTO SEGMENT BUFFER

  function READ_CHARS(SEG : SEG_PTR;
                      WHERE,COUNT : CHAR_COUNT) return STRING;
  -- READS COUNT CHARACTERS STARTING AT WHERE

  function READ_CHARACTER_COUNT(SEG : SEG_PTR) return CHAR_COUNT;
  -- RETURNS THE NUMBER OF CHARACTERS IN THE BUFFER

  procedure SET_TIME(SEG : SEG_PTR; HACK : DATE_TIME);
  function READ_TIME(SEG : SEG_PTR) return DATE_TIME;
  -- SET AND READ TIME HACK OF SEGMENT

  procedure SET_PART(SEG : SEG_PTR; PART : PART_NAME);
  function READ_PART(SEG : SEG_PTR) return PART_NAME;
  -- SET AND READ THE PART_NAME FIELD OF THE SEGMENT

  procedure SET_EASN(SEG : SEG_PTR; SERNO : EXT_SERIAL_NO);
  function READ_EASN(SEG : SEG_PTR) return EXT_SERIAL_NO;
  -- SET AND READ EXTENDED SERIAL NUMBER

  procedure SET_SEGMENT_NUMBER(SEG : SEG_PTR;
                               NUMBER : SEGMENT_NUMBER);
```

```
   function READ_SEGMENT_NUMBER(SEG : SEG_PTR) return SEGMENT_NUMBER;
   -- SET AND READ SEGMENT SEQUENCE FIELD

   procedure SET_CLASS(SEG : SEG_PTR; CLASS : SECURITY_CLASSIFICATION);
   function READ_CLASS(SEG : SEG_PTR) return SECURITY_CLASSIFICATION;
   -- SET AND READ CLASSIFICATION

   procedure LINK_SEGMENTS(SEG, PRIOR_SEG : SEG_PTR);
   -- LINKS SEG INTO A SEGMENT CHAIN
   -- SETS FORWARD POINTER IN PRIOR SEGMENT TO SEG
   -- SETS BACKWARD POINTER IN SEG TO PRIOR_SEG
   -- SETS FORWARD POINTER IN SEG TO NULL

   function NEXT_SEGMENT(SEG : SEG_PTR) return SEG_PTR;
   function PRIOR_SEGMENT(SEG : SEG_PTR) return SEG_PTR;
   -- READ NEXT AND PRIOR SEGMENT POINTERS

   procedure FREE_SEGS(SEG : SEG_PTR) ;
   -- FREES AN ENTIRE CHAIN OF SEGMENTS

   function GET is new GET_INTRANSIT(SEGMENT,SEG_PTR);
   procedure FREE is new FREE_INTRANSIT(SEGMENT,SEG_PTR);
   -- GET AND FREE INTRANSIT STORAGE FOR SEGMENTS
   -- SEE MANAGE_STORAGE FOR DETAILS

private


   type SEGMENT is
     record
       BACK_SEG            : SEG_PTR                    :=null;
       CLASS               : SECURITY_CLASSIFICATION;
       EASN                : EXT_SERIAL_NO;
       FWD_SEG             : SEG_PTR                    :=null;
       NUMBER_CHARACTERS   : CHAR_COUNT                 := 0;
       PART                : PART_NAME;
       SEG_NO              : SEGMENT_NUMBER;
       TEXT                : STRING(1..80);
       TIME                : DATE_TIME;
     end record;

end SEGMENT_OPS;
```

### 3.1.2 Message Queueing

```
with MESSAGE_OPS;
use MESSAGE_OPS;

package MANAGE_QUEUES is
  ------------------------------------------------
  --  NAME:  MANAGE_QUEUES
  --  PURPOSE:  Task that manages messages after reception while they are
  --       awaiting processing for routing.
  ------------------------------------------------

  task MANAGE_MESSAGE_QUEUES is
    entry QUEUE_NORMAL ( MESSAGE: MSGID );
    entry QUEUE_SERVICE ( MESSAGE: MSGID );
    entry QUEUE_REINTROD ( MESSAGE: MSGID );
    entry DEQUEUE_OVERFLOW ( MESSAGE: out MSGID );
    entry RETRIEVE_FOR_PROCESS ( PRECEDENCE ) ( MESSAGE: out MSGID );
  end MANAGE_MESSAGE_QUEUES;

end MANAGE_QUEUES;
```

```
with MESSAGE_OPS, GLOBAL_TYPES, LINE_TBL_OPS, QUEUE;
use MESSAGE_OPS, GLOBAL_TYPES;

package MANAGE_TRANSLATED_QUEUES is
  -------------------------------------------------
  --  NAME:  MANAGE_TRANSLATED_QUEUES
  --  PURPOSE:  Tasks that manage messages after translation while they ar
  --      awaiting line availabilty.
  -------------------------------------------------

  task type MANAGE_OUTGOING is
    entry INIT ( LOG_ID: LOGICAL_LINE );
    entry SET_PHYS_LINES ( PHYS_LINES: LINE_TBL_OPS.PHYS_PTR );
    entry QUEUE_TRANSLATED ( THIS_LINE: LOGICAL_LINE; MESSAGE: MSGID );
    entry QUEUE_REINTROD ( THIS_LINE: LOGICAL_LINE; MESSAGE: MSGID );
    entry LINE_READY ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
    entry KILL_NOW ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
    entry KILL_LINE ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
    entry KILL_ON_EMPTY ( THIS_LINE: LINE_TBL_OPS.PHYSICAL_LINE );
    entry INTERCEPT ( THIS_LINE: LOGICAL_LINE; PREC: PRECEDENCE;
        MEDIUM: LINE_TBL_OPS.MEDIA );
    entry REMOVE_INTERCEPT ( THIS_LINE: LOGICAL_LINE; PREC: PRECEDENCE;
        MEDIUM: LINE_TBL_OPS.MEDIA );
  end MANAGE_OUTGOING;

  type MANAGER is access MANAGE_OUTGOING;

  task MANAGE_INTERCEPT is
    entry QUEUE_ONE ( MESSAGE: MSGID );
    entry QUEUE_MANY ( LIST: QUEUE.HEAD );
    entry DEQUEUE ( MESSAGE: MSGID );
  end MANAGE_INTERCEPT;

  task MOVE_INTERCEPT is
    entry START_INTERCEPT_OUT;
    entry STOP_INTERCEPT_OUT;
    entry START_REINTRO_OF_INTERCEPT;
    entry STOP_REINTRO_OF_INTERCEPT;
  end MOVE_INTERCEPT;

end MANAGE_TRANSLATED_QUEUES;
```

```
with GLOBAL_TYPES, MESSAGE_OPS, SEGMENT_OPS; use GLOBAL_TYPES,
    MESSAGE_OPS, SEGMENT_OPS;

package MESSAGE_IO is
-------------------------------------------------
--  NAME:  MESSAGE_IO
--  PURPOSE:  Input/output subprograms used to read/write messages to/from
--       files ( ie., OVERFLOW, INTERCEPT)
-------------------------------------------------
  procedure READ(FILE_NAME:STRING; MESSAGE:out MSGID);
  procedure WRITE(FILE_NAME:STRING; MESSAGE:MSGID);
private
  type KIND is (VERSION_HEADER, PART_HEADER, SEGMENT);
  type BLOCK(PART_KIND:KIND :=SEGMENT) is
    record
      case PART_KIND is
        when VERSION_HEADER =>
          VERSION:VERSION_HEADER;
        when PART_HEADER =>
          PART_HEAD:PART_HEADER;
        when SEGMENT =>
          SEG:SEGMENT;
      end case;
    end record;
  package BLOCK_IO is new INPUT_OUTPUT(BLOCK);
  type IN_FILE is new BLOCK_IO.IN_FILE;
  type OUT_FILE is new BLOCK_IO.OUT_FILE;

end MESSAGE_IO;
```

```
with GLOBAL_TYPES,MESSAGE_OPS; use GLOBAL_TYPES,MESSAGE_OPS;

package QUEUE is
  ------------------------------------------------------
  --  NAME: QUEUE
  --  PURPOSE: Encapsulates the queue mechanisms used to hold the messages
  --      during waiting stages as they pass thru the switch.
  --  PROGRAMMER: SLN
  --  DATE: 18 MAY 82
  ------------------------------------------------------
  type HEAD is private;
  type SET is array ( MESSAGE_CAT, PRECEDENCE ) of HEAD;
  procedure ON_FRONT( QUEUE: in out HEAD; MESSAGE: in MSGID );
    -- Place a message properly in a queue, ( when it is being
    --      reintroduced as from overflow or intercept ).
  procedure ON_BACK( QUEUE: in out HEAD; MESSAGE: in MSGID );
    -- Place a message properly on a queue.
  procedure FROM_FRONT( QUEUE: in out HEAD; MESSAGE: out MSGID );
    -- Retreive a message from the front of a queue for normal
    --      processing.
  procedure FROM_BACK( QUEUE: in out HEAD; MESSAGE: out MSGID );
    -- Retreive a message from the back of a queue for overflow.
  function IS_NOT_EMPTY( QUEUE: in HEAD ) return BOOLEAN;
    -- Test the empty state of a queue
  procedure SPLIT( MEDIUM: in MEDIA; QUEUE: in out HEAD;
    -- Segregate messages of a particular media from a queue into a
    --      seperate list.
      LIST: out HEAD );
  procedure MERGE( MASTER: in out HEAD; SLAVE: in HEAD );
    -- Combine two queues into one.
  FOR_PROCESSING: SET;
  FOR_OUTGOING: array ( LOGICAL_LINES ) of SET;
  FOR_INTERCEPT: SET;

private
  type ITEM;
  type ITEM_PTR is access ITEM;
  type ITEM is
    record
      NEXT,PRIOR: ITEM_PTR;
      MESSAGE: MSGID;
      TOR: DATE_TIME;
    end record;
  type HEAD is
    record
      FIRST,LAST: ITEM_PTR := null;
      COUNT: INTEGER :=0;
    end record;
end QUEUE;
```

### 3.1.3 Message Routing

PROCESS_MSG

```
with MESSAGE_OPS,RI_OPS,LINE_TBL_OPS;use MESSAGE_OPS,RI_OPS,
    LINE_TBL_OPS;

package PROCESS_MSG is
  ------------------------------------------------------------------
  -- NAME: PROCESS_MSG
  -- PURPOSE: contains the task required to route and translate
  --          a message.  (translations required depend on the
  --          type of exchange required.)
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------
  task PROCESS_MESSAGE;
end PROCESS_MSG;
```

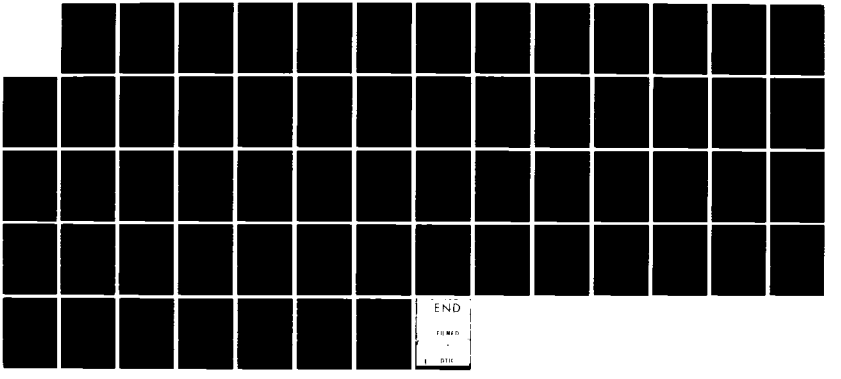
### 3.1.4 Message Output

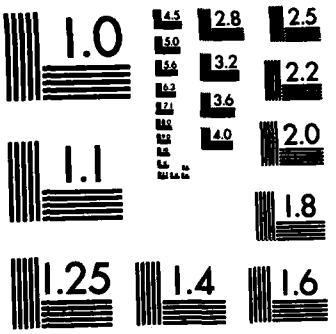
```
-- JUNE 22 10:35    A.M. -- PBD

with GLOBAL_TYPES, MESSAGE_OPS, TIME_OPS, HISTORY_OPS,
    LINE_TBL_OPS, RI_OPS, HARDWARE_CLOCK, CONVERT_TO_ITA, TASKS,
    SERVICE_MESSAGE_OPS;
use GLOBAL_TYPES, MESSAGE_OPS, TIME_OPS, HISTORY_OPS,
    LINE_TBL_OPS, RI_OPS, HARDWARE_CLOCK, CONVERT_TO_ITA,
    SERVIVE_MESSAGE_OPS;

package PHYSICAL_PORT is
  ------------------------------------------------------------------
  -- NAME: PHYSICAL_PORT
  -- PURPOSE: contains the operations and data definitions dealing with
  --          operations of the physical output ports, not including
  --          operator output.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type LOG_REQUEST (LOG_TYPE : LOG_ENTRY) is
    record
      BLOCKS   : NUM_BLOCKS;
      LINE     : PHYSICAL_LINE;
      MSG      : MSGID;
      SEQ_NUM  : CSN;
      case LOG_TYPE is
        when CANCEL_OUT | CANTRAN_OUT =>
          CAN_REASON  : CANCEL_REASON;
        when SCRUB =>
          SCR_REASON : SCRUB_REASON;
        when REJECT =>
          REJ_REASON : REJECT_REASON;
        when others =>
          null;
      end case;
    end record;

  type MSG_STAT is (NORMAL_ACC, COMPLETED_ACC, REJ);

  task type OUTPUT_MESSAGE is
    entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
    entry SEND_MESSAGE(MSG : MSGID); -- called from queue task
    entry PREEMPT_MESSAGE(NEW_MSG : MSGID;
                          OLD_MSG : out MSGID;
                          STATUS : out MSG_STAT);  -- called from queue
    entry FINISHED_SENDING;  -- called by send
    entry PREEMPT_CHECK(MSG : out MSGID);  -- called by send
    entry GET_MESSAGE(MSG : out MSGID); -- called by send
  end OUTPUT_MESSAGE;
```

```
    task type SEND_MODE_II_IV is
      entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
    end SEND_MODE_II_IV;

    task type SEND_MODE_V is
      entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
    end SEND_MODE_V;

    task type SEND_MODE_I is
      entry INIT(LINE : PHYSICAL_LINE); -- called by run switch
    end SEND_MODE_I;

    task type CONTROL_CHARACTER is
      entry INIT(P_LINE : PHYSICAL_LINE); -- called by run switch
      entry PUT(CH : CHARACTER);  -- called by receiver
      entry GET(CH : out CHARACTER);  -- called by send
    end CONTROL_CHARACTER;

    task type DECOUPLE is
      entry LOG(REQ : LOG_REQUEST);  -- called by send
    end DECOUPLE;

    type DECOUPLE_ACC is access DECOUPLE;

    task type THROTTLE_INPUT is
      entry INIT(LINE : PHYSICAL_LINE);  -- called by run switch
      entry START_THROTTLE;
      entry STOP_THROTTLE;
    end THROTTLE_INPUT;

end PHYSICAL_PORT;
```

AD-A123 307 LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MES..(U) GENERAL DYNAMICS FORT WORTH TX DATA SYSTEMS DIV 09 NOV 82 DAAK80-81-C-0108 2/2

UNCLASSIFIED F/G 17/2 NL





### 3.1.5 Manage Intransit, Manage Overflow

```
package MANAGE_INTRANSIT is
  ------------------------------------------------------------------
  -- NAME: MANAGE_INTRANSIT
  -- PURPOSE: contains procedures and tasks to manage the intransit
  --          storage memory resource.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type MEM_SIZE is INTEGER;
  type THRESHOLD is range 0..100;

  procedure SET_THRESHOLD(LOWER:THRESHOLD := 60;
      UPPER:THRESHOLD := 70);
  -- SETS LOWER AND UPPER THRESHOLD AS SPECIFIED IN CALL
  -- MIDDLE THRESHOLD IS SET TO THE AVERAGE OF LOWER AND UPPER

  procedure READ_THRESHOLD(LOWER,MIDDLE,UPPER: out THRESHOLD);
  -- READS CURRENT THRESHOLD SETTINGS

  task CALCULATE_THRESHOLD is
    -- CALCULATES THRESHOLD VALUES AND INITIATES OVERFLOW
       -- ACTIONS
    entry GET(BITS:MEM_SIZE);
    -- USED WHEN GETTING STORAGE
    entry PUT(BITS:MEM_ISZE);
    -- USED WHEN RETURNING STORAGE TO INTRANSIT
  end CALCULATE_THRESHOLD;

end MANAGE_INTRANSIT;
```

```
with MESSAGE_OPS,MESSAGE_IO; use MESSAGE_OPS,MESSAGE_IO;

package MANAGE_OVERFLOW is
  ------------------------------------------------------------------
  -- NAME: MANAGE_OVERFLOW
  -- PURPOSE: contains the procedures to handle the overflow of
  --          intransit storage
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  task OVERFLOW_DRIVER is
    entry START_OVERFLOW;
    entry STOP_OVERFLOW;
    entry START_REENTRY;
    entry STOP_REENTRY;
  end OVERFLOW_DRIVER;

end MANAGE_OVERFLOW;
```

### 3.1.6 Support Routines

```
package ASN is
  ------------------------------------------------------------------
  -- NAME: ASN
  -- PURPOSE: provides the definition of the type SER_NO_TYPE and
  --          a task to provide sequential serial numbers.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 14, 1982
  ------------------------------------------------------------------
  type SER_NO_TYPE is private;

  task MANAGE_ASN is
    entry SET(START:SER_NO_TYPE);
    entry GET(NO:out SER_NO_TYPE);
  end MANAGE_ASN;

private
  type SER_NO_TYPE is range 1..1_000_000;
end ASN;
```

```
with GLOBAL_TYPES;use GLOBAL_TYPES;

package AUDIT is
  ------------------------------------------------------------------
  -- NAME: AUDIT
  -- PURPOSE: contains data definitions and operations required to
  --          audit the history and produce lists of current messages
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type AUDIT_LIST_NODE;
  type AUDIT_LIST_PTR is access AUDIT_LIST_NODE;
  type AUDIT_LIST_NODE is
    record
      MESSAGE:EASN;
      NEXT:AUDIT_LIST_PTR;
    end record;
  type AUDIT_RECORD is
    record
      NUM_INTRANSIT:INTEGER := 0;
      NUM_IN_OVERFLOW:INTEGER := 0;
      NUM_IN_INTERCEPT:INTEGER := 0;
      AUDIT_LIST:AUDIT_LIST_PTR;
    end record;
  CURRENT_AUDIT:AUDIT_RECORD;
  -- RUNNING AUDIT FIGURES

  function AUDIT_INTRANSIT return AUDIT_RECORD;
  -- READS LOG AND PRODUCES A NEW AUDIT WITH AT LIST OF
      -- INTRANSIT MESSAGES

  type OVERFLOW_RECORD is new AUDIT_RECORD;
  function AUDIT_OVERFLOW return OVERFLOW_RECORD;
  -- READS LOG & PRODUCES AN AUDIT WITH A LIST OF OVERFLOW MESSAGES

  type INTERCEPT_RECORD is new AUDIT_RECORD;
  function AUDIT_INTERCEPT return INTERCEPT_RECORD;
  -- READS LOG AND PRODUCES AN AUDIT WITH A LIST OF INTERCEPTED
      -- MESSAGES

  function EQUAL(A,B:AUDIT_RECORD)return BOOLEAN;
  -- COMPARES TWO AUDITS FOR SAME NUMBERS AND SAME MESSAGES ON
  -- LISTS.  THE ORDER OF LISTS ARE NOT SIGNIFICANT.

  procedure WRITE_AUDIT(OUTPUT_AUDIT:AUDIT_RECORD);
  -- WRITES AUDIT RECORD ON HISTORY TAPES

end AUDIT;
```

```
package BYTE_DEF is

------------------------------------------------------------------------
--Name:          BYTE_DEF
--Purpose:       to define bytes, words, and long words
--Programmer:    Brian G. Sharpe
--Date:          11 June 1982
--Description:
--     Bytes, words, and long words are defined to have 8, 16, and 32
--     bits, respectively.  Also defined are conversions between the
--     different types.
------------------------------------------------------------------------

  BYTE_SIZE      : constant  INTEGER := 8;
  MAX_BYTE_VALUE: constant  INTEGER := 2 ** BYTE_SIZE  -  1;
  subtype BYTE is INTEGER 0..MAX_BYTE_VALUE;

  WORD_SIZE      : constant  INTEGER := 2 * BYTE_SIZE;
  MAX_WORD_VALUE: constant  INTEGER := 2 ** WORD_SIZE  -  1;
  subtype WORD is INTEGER 0..MAX_WORD_VALUE;

  LONG_WORD_SIZE      : constant  INTEGER := 2 * WORD_SIZE;
  MAX_LONG_WORD_VALUE: constant  INTEGER := 2 ** LONG_WORD_SIZE  -  1;
  subtype LONG_WORD is INTEGER 0..MAX_LONG_WORD_VALUE;

  -- representation specifications for the sizes of the above types
  for  BYTE'SIZE use BYTE_SIZE;
  for  WORD'SIZE use WORD_SIZE;
  for  LONG_WORD'SIZE use LONG_WORD_SIZE;


  --merge two bytes into a word
  function BYTES_TO_WORD(
                MST_SIG_BYTE,
                LST_SIG_BYTE : in BYTE )    return WORD;

  --merge two words into a long word
  function WORDS_TO_LONG_WORD(
                MST_SIG_WORD,
                LST_SIG_WORD : in WORD )    return LONG_WORD;

end BYTE_DEF;


package body BYTE_DEF is
```

```
package CONTROL is
  ------------------------------------------------------------------
  -- NAME: CONTROL
  -- PURPOSE: provides a task type for controlling access to
  --          a shared resource.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 5, 1982
  ------------------------------------------------------------------

  task type CONTROL_ACCESS is
    entry START_READ;
    entry FINISHED_READ;
    entry START_WRITE;
    entry FINISHED_WRITE;
  end CONTROL_ACCESS;

end CONTROL;
```

```
with   INTERFACE_TO_8254 , BYTE_DEF ;
use    INTERFACE_TO_8254 , BYTE_DEF ;

package HARDWARE_CLOCK is

------------------------------------------------------------------------
--Name:          HARDWARE_CLOCK
--Purpose:       to implement a hardware, binary counter clock
--Programmer:    Brian G. Sharpe
--Date:          8 June 1982
--Description:
--  This package implements a hardware clock using a binary counter.
--  It is implemented when one needs a time DELTA which is less than
--  DURATION'DELTA.
--
--                  !!!!!!!!!!! NOTE !!!!!!!!!!
--  This is useful for elapsed time only; it is not an absolute
--  (i.e., time of day) clock.
--
-- Since the elapsed timer is implemented in hardware with binary clock,
-- it will "wrap around" from "COUNTER_RANGE_MAX" to 0.
--
-- Thus, the user must ensure that the largest possible elapsed time will
--. be less than the time to pass once through the counter.
------------------------------------------------------------------------

  -- package declaration section:

  type HARDWARE_CLOCK_TIME is private;

  -- Subprogram to start the clock chip running in the proper configuration
  procedure INIT_HARDWARE_CLOCK;

  function CONVERT_TO_SECONDS( TIME_IN_HCT : HARDWARE_CLOCK_TIME )
                          return float;

  -- Determines the elapsed time between inputs
  function ELAPSED_TIME( START_TIME, FINAL_TIME : in HARDWARE_CLOCK_TIME );

  -- Function to read the hardware clock
  function HARDWARE_CLOCK_READING  return HARDWARE_CLOCK_TIME;


  private  -- section
    CLOCK_SIZE : constant := LONG_WORD_SIZE ;     -- # bits in counter clock
    COUNTER_RANGE_MAX : constant :=  MAX-LONG_WORD_VALUE  ;
    CLOCK_CHANNEL_LOW_WORD  is constant CHANNEL_0;
    CLOCK_CHANNEL_HIGH_WORD is constant CHANNEL_1;
    USE_CHANNEL : SELECT_CHANNEL_ARRAY := ( true, true, false );
```

```
-- APRIL 07 1:24:00 -- PBD

with RI_OPS, LINE_TBL_OPS, TIME_OPS, SEGMENT_OPS, GLOBAL_TYPES;
use RI_OPS, LINE_TBL_OPS, TIME_OPS, SEGMENT_OPS, GLOBAL_TYPES;

package HISTORY_OPS is
  type ENTRY_TYPE is (LOG, RI_SET, SEGMENT);
  type NUM_RIS is range 0..50;
  type LOG_ENTRY is (SOM_IN, EOM_IN, REJECT, CANCEL_IN,
      CANTRAN_IN, SOM_OUT, EOM_OUT, CANCEL_OUT, CANTRAN_OUT,
      SCRUB, OUT_TO_OVFL, IN_FROM_OVFL, OUT_TO_INT, IN_FROM_INT,
      SVC_GEN, VERSION_OUT, VERSION_IN);
  type REJECT_REASON is (INVALID_HDR, INVALID_RI, INVALID_SECURITY,
      SECURITY_MISMATCH);
  type CANCEL_REASON is (PREMPTED, REJ_BY_RCVR);
  type SCRUB_REASON is (OPERATOR);
  type NUM_BLOCKS is range 0..50;
  type RI_ARRAY is array (INTEGER range <>) of RI_STRING;
  type HISTORY_ENTRY (ENT:ENTRY_TYPE:=SEGMENT;
      RI_COUNT:NUM_RIS:=0; LOG_TYPE:LOG_ENTRY:=SOM_IN) is
    record
      case ENT is
        when LOG =>
          CHANNEL:CHANNEL_DES;
          SEQ_NUM:CSN;
          HACK:DATE_TIME;
          MODE:CHANNEL_MODE;
          BLOCK_COUNT:NUM_BLOCKS:=0;
          MCB:BOOLEAN:=FALSE;
          HEADER_SEGS:NUM_BLOCKS:=0;
          RIS:BOOLEAN:=FALSE;
          case LOG_TYPE is
            when REJECT  >
              RE _REASON:REJECT_REASON;
            when CANCEL_OUT =>
              CAN_REASON:CANCEL_REASON;
            when SCRUB =>
              SCR_REASON:SCRUB_REASON;
            when others =>
              null;
          end case;
        when RI_SET =>
          RIS:RI_ARRAY (1..RI_COUNT);
        when SEGMENT =>
          SEG_BUF:SEGMENT;
      end case;
    end record;
  type ENTRY_SET is array (INTEGER range <>) of HISTORY_ENTRY;
  type HIST_STATUS is (OK, END_OF_MEDIA, ERROR);
  type MESSAGE_LIST is array (INTEGER range <>) of EASN;

  task ENTER_HIST is
    entry WRITE(REC:ENTRY_SET; STATUS: out HIST_STATUS);
  end ENTER_HIST;
  -- USED TO WRITE A RECORD OR SET OF RECORDS ON THE HISTORY
```

```
with ASN;  use ASN;

package GLOBAL_TYPES is
  ------------------------------------------------------------------
  -- NAME:        GLOBAL_TYPES
  -- PURPOSE:     This package contains the types global to the
  --              entire software project.
  -- PROGRAMMER: Paul Dobbs
  -- DATE:        May 17, 1982
  ------------------------------------------------------------------

  subtype   CHANNEL_DES     is STRING(1..3);
  type      CHANNEL_MODE    is range 1..5;
  type      CHAR_SET_TYPE   is ( ANY, ASC, ITA );
  type      CSN             is range 0..999;

  type  EXT_SERIAL_NO  is
    record
      ASN           : SER_NO_TYPE;
      EXTENSION     : INTEGER range 0..100;
    end record;

  type  JULIAN_DATE    is range 1..366;

  type  DATE_TIME      is
    record
      DATE : JULIAN_DATE;
      TIME : DURATION;
    end record;

  type  LOGICAL_LINE   is range 0..50;
  type  PART_NAME      is ( MCB, HEADER, MSG_BODY, TRAILER );
  type  PRECEDENCE     is
                     ( ROUTINE,          -- 'R'
                       PRIORITY,         -- 'P'
                       IMMEDIATE,        -- 'O'
                       FLASH,            -- 'Z'
                       ECP,              -- 'Y'
                       CRITIC      );    -- 'W'
  type  SECURITY_CLASSIFICATION
                       is
                     ( UNCLASS,          -- 'U'
                       EFTO,             -- 'E'
                       RESTRICTED,       -- 'R'
                       CONFIDENTIAL,     -- 'C'
                       SECRET,           -- 'S'
                       TOP_SECRET,       -- 'T'
                       SPECAT,           -- 'A'
                       DSSCS   );        -- 'M'
```

```
end GLOBAL_TYPES;
```

```
with RI_OPS, LINE_TBL_OPS, SEGMENT_OPS, GLOBAL_TYPES, MESSAGE_OPS;
use RI_OPS, LINE_TBL_OPS, SEGMENT_OPS, GLOBAL_TYPES, MESSAGE_OPS;

package HISTORY_OPS is
  ------------------------------------------------------------------
  -- NAME: HISTORY_OPS
  -- PURPOSE: contains data definitions and operations pertaining to
  --          the history files (log and journal).
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type CANCEL_REASON is (PREMPTED, REJ_BY_RCVR);
  type ENTRY_TYPE is (LOG, RI_SET, SEGMENT);
  type HIST_STATUS is (OK, END_OF_MEDIA, ERROR);
  type LOG_ENTRY is (SOM_IN, EOM_IN, REJECT, CANCEL_IN, CANTRAN_IN,
                     SOM_OUT, EOM_OUT, CANCEL_OUT, CANTRAN_OUT,
                     SCRUB, OUT_TO_OVFL, IN_FROM_OVFL, OUT_TO_INT,
                     IN_FROM_INT, SVC_GEN, VERSION_OUT, VERSION_IN);
  type MESSAGE_LIST is array (INTEGER range <>) of EXT_SERIAL_NO;
  type NUM_BLOCKS is range 0..50;
  type REJECT_REASON is (INVALID_HDR, INVALID_RI,
                         INVALID_SECURITY, SECURITY_MISMATCH);
  type RI_ARRAY is array (NUM_RIS range <>) of RI_STRING;
  type SCRUB_REASON is (OPERATOR);

  type HISTORY_ENTRY (ENT : ENTRY_TYPE := SEGMENT;
                      RI_COUNT : NUM_RIS := 0;
                      LOG_TYPE : LOG_ENTRY := SOM_IN) is
    record
      case ENT is
        when LOG =>
          BLOCK_COUNT : NUM_BLOCKS;    -- Blocks transmitted or received
          CHANNEL     : CHANNEL_DES;   -- Channel designator
          HACK        : DATE_TIME;     -- Time of event
          HEADER_SEGS : NUM_BLOCKS;    -- Number of header segments which
                                       --    are present in the log entry
                                       -- May be zero
          MCB         : BOOLEAN;       -- Whether an MCB segment
                                       --    is present
          MODE        : CHANNEL_MODE;  -- Mode of the channel
          RIS         : BOOLEAN;       -- Whether an RI_set is included
          SEQ_NUM     : CSN;           -- CSN for asynch only
          SER_NO      : EXT_SERIAL_NO; -- Message extended serial number
          case LOG_TYPE is             -- Reasons, where applicable
            when REJECT =>
              REJ_REASON : REJECT_REASON;
            when CANCEL_OUT | CANTRAN_OUT =>
              CAN_REASON : CANCEL_REASON;
            when SCRUB =>
```

```
            SCR_REASON : SCRUB_REASON;
          when others =>
            null;
        end case;
      when RI_SET =>
        RI_S:RI_ARRAY (1..RI_COUNT);
      when SEGMENT =>
        SEG_BUF : SEGMENT_OPS.SEGMENT;
    end case;
  end record;

type ENTRY_SET is array (INTEGER range <>) of HISTORY_ENTRY;
-- An ENTRY_SET consists of a HISTORY_ENTRY of type LOG followed by:
    -- An optional RI_SET
    -- An optional MCB SEGMENT
    -- 0 or more header SEGMENTs
-- This ordering MUST NOT be violated

task ENTER_HIST is
  entry WRITE(REC : ENTRY_SET;
              STATUS: out HIST_STATUS);
end ENTER_HIST;
-- USED TO WRITE A RECORD OR SET OF RECORDS ON THE HISTORY

procedure SEARCH_LOG(SER_NO : EXT_SERIAL_NO;
                     REC : out ENTRY_SET;
                     STATUS : out HIST_STATUS);
-- SEARCHES LOG FOR THE NEXT LOG ENTRY FOR A PARTICULAR EASN

procedure READ_LOG(REC : out ENTRY_SET;
                   STATUS : out HIST_STATUS);
-- READS THE NEXT LOG ENTRY -- MAY SKIP REFERENCE ENTRIES

procedure REWIND_LOG;
-- RESETS LOG TO THE BEGINNING OF THE SHORT HISTORY

procedure INIT_HISTORY;
-- STARTS A FRESH HISTORY

procedure NEW_DAY;
-- CLOSES PREVIOUS DAY'S HISTORY AND STARTS NEW DAY
    -- CLOSES OLD DAY'S TAPE
    -- WRITES CURRENT RUNNING AUDIT ON END OF OLD DAY
    -- WRITES CURRENT RUNNING AUDIT ON START OF NEW DAY
    -- OPENS NEW DAY'S TAPE
    -- AUDITS OLD DAY TAPE AND COMPARES WITH RUNNING AUDIT
    -- PRINTS STATISTICAL SUMMARY OF DAY'S ACTIVITIES

procedure CLOSE_HISTORY;
-- CLOSES HISTORY WITHOUT STARTING A NEW ONE
```

```
        -- CLOSES OLD DAY'S TAPE
        -- WRITES CURRENT RUNNING AUDIT TO END OF TAPE
        -- AUDITS TAPE AND COMPARES WITH RUNNING AUDIT
        -- PRINTS A STATISTICAL SUMMARY OF DAY'S ACTIVITIES

    procedure RE_ENTER_MESSAGES(LIST : MESSAGE_LIST;
                                STATUS : out HIST_STATUS;
                                NOT_FOUND : out MESSAGE_LIST);
    -- READS A LIST OF MESSAGES FROM HISTORY INTO INTRANSIT AND
        -- PLACES ON QUEUE
    -- RETURNS A LIST OF MESSAGES NOT RECOVERED

    procedure READ_MESSAGE(SER_NO : EXT_SERIAL_NO;
                           PTR : out MSGID;
                           STATUS : out HIST_STATUS);
    -- READS A MESSAGE INTO INTRANSIT AND RETURNS A MSGID

end HISTORY_OPS;
```

```
with BYTE_DEF ; use BYTE_DEF ;

package INTERFACE_TO_8254 is

------------------------------------------------------------------------
--Name:          INTERFACE_TO_8254
--Purpose:       interface to the Intel 8254 counter/timer IC
--Programmer:    Brian G. Sharpe
--Date:          14 June 1982
--Description:
--     This package provides all of the necessary interfaces to the Intel
--     8254 counter/timer IC.  The necessary byte formats are defined as
--     types and subprograms are provided in order to provide information
--     hiding.
--     This package is not complete; basically, only those subprograms
--     that are required have been defined and coded.
--
--               ***********  NOTE  **********
--       For sake of simplicity, memory-mapped I/O is assumed to be used.
--       Use of non-memory-mapped I/O will require something other than
--       SHARED_VARIABLE_UPDATE in order to read and write to the hardware.
------------------------------------------------------------------------

  -- declaration section:

  type TYPE_OF_COUNTING_FORMAT is ( BINARY, BCD );

  -- The following modes determine the operating modes of the
  -- programmable 8254.
  type PERMISSABLE_MODES is
            ( MODE_0,          -- interrupt on terminal count
              MODE_1,          -- hardware retriggerable one-shot
              MODE_2,          -- rate generator
              MODE_3,          -- square wave mode
              MODE_4,          -- software triggered strobe
              MODE_5   );      -- hardware trtiggered strobe (retriggerable

  type PERMISSABLE_CHANNELS_OR_READ_BACK is
            ( CHANNEL_0,       -- select counter channel 0
              CHANNEL_1,
              CHANNEL_2,
              READ_BACK    ); -- select read-back command mode

  subtype PERMISSABLE_CHANNELS is PERMISSABLE_CHANNELS_OR_READ_BACK
                                  range CHANNEL_0..CHANNEL_2;


  type CHANNEL_WORD_ARRAY is array( CHANNEL_0 .. CHANNEL_2 ) of WORD;
  type SELECT_CHANNEL_ARRAY is array( CHANNEL_0 .. CHANNEL_2 ) of BOOLEAN;
```

```
    -- read any combination of the three 16-bit channels
    function READ_MULTIPLE_CHANNELS
                 ( READ_CHNL : in SELECT_CHANNEL_ARRAY )
                   return CHANNEL_WORD_ARRAY;

    -- get the clock frequencies of the given channel
    function CHANNEL_CLOCK_FREQ ( CHANNEL_NO : in PERMISSABLE_CHANNELS )
        return FLOAT;

    -- set the counting format (binary or BCD) of the given channel
    procedure SET_COUNTING_FORMAT
                ( CHANNEL_NO      : in PERMISSABLE_CHANNELS;
                  COUNTING_FORMAT : in TYPE_OF_COUNTING_FORMATS );

    -- set the operating mode of the given channel
    procedure SET_OPERATING_MODE
                ( CHANNEL_NO       : in PERMISSABLE_CHANNELS;
                  OPERATING_FORMAT : in PERMISSABLE_MODES );


end INTERFACE_TO_8254;
```

MESSAGE_OPS

```
with GLOBAL_TYPES,SEGMENT_OPS,MANAGE_STORAGE;
use GLOBAL_TYPES,SEGMENT_OPS,MANAGE_STORAGE;

package MESSAGE_OPS is
  ------------------------------------------------------------------
  -- NAME: MESSAGE_OPS
  -- PURPOSE: contains data definitions and operation definitions
  --          dealing with the message.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type BLOCK_COUNT is new STRING(1..3);
  type ERR_STAT is (OK,END_OF_TEXT,LINK_ERROR,OTHER_ERROR);
  type FORMAT is (ACP,JANAP);
  type MESSAGE_CATEGORY is (NORMAL,SERVICE);
  type NUM_RIS is range 0..50;
  type PART_HEADER is private;
  type PART_PTR is access PART_HEADER;
  type POSITION is limited private;
  type RI_STAT is (OK,LAST_RI,ERROR);
  type SEG_ARRAY is array(NATURAL range <>) of SEGMENT;
  type VERSION_HEADER is private;
  type MSGID is access VERSION_HEADER;

  function ESTABLISH_VERSION(EASN : EXT_SERIAL_NO) return MSGID;
  -- ESTABLISH A MESSAGE VERSION HEADER
  -- HEADER HAS NO PARTS

  procedure OPEN_FOR_READ(MESSAGE : MSGID;
                          START_WITH : PART_NAME;
                          POS:out POSITION);
  -- ESTABLISH INITIAL POSITION FOR READING FROM A MESSAGE

  procedure READ_CONTINUOUS(MESSAGE : MSGID;
                            COUNT : in out CHAR_COUNT;
                            WHERE_FROM : in out POSITION;
                            TEXT : out STRING;
                            STATUS : out ERR_STAT);
  procedure READ_FROM_PART(MESSAGE : MSGID;
                           COUNT : in out CHAR_COUNT;
                           WHERE_FROM : in out POSITION;
                           TEXT : out STRING;
                           STATUS : out ERR_STAT);
  -- READ COUNT CHARACTERS STARTING FROM WHERE_FROM
  -- CHARACTERS ARE RETURNED IN TEXT
  -- IF STATUS IS END_OF_TEXT, COUNT WILL REFLECT ACTUAL NUMBER
      -- OF CHARACTERS READ
  -- CHECK FOR LINKING ERRORS AND RETURNS ERROR IF SEGMENTS
      -- ARE NOT LINKED IN A CONSISTENT FASHION
```

```
procedure ATTACH_SEGMENT(MESSAGE : MSGID; SEG : SEG_PTR);
-- ATTACHES A SEGMENT TO THE MESSAGE IN THE PART
    -- SPECIFIED IN THE SEGMENT
-- IF NO PART_HEADER EXISTS, ONE WILL BE CREATED

procedure WRITE_TO_PART(MESSAGE : MSGID;
                        PART : PART_NAME;
                        TEXT : STRING);
-- WRITES TEXT TO THE MESSAGE AND PART SPECIFIED
-- CREATES PART_HEADERS AND SEGMENTS AS REQUIRED

procedure ATTACH_PART(MESSAGE, OLD_MESSAGE : MSGID;
                      PART : PART_NAME);
-- MAKES ADDITIONAL USE OF PART IN OLD_MESSAGE IN THE CURRENT
    -- MESSAGE
-- INCREMENTS USER_COUNT IN PART_HEADER

procedure FREE_PART(MESSAGE : MSGID; PART : PART_NAME);
-- DETACHES AND FREES (IF NO OTHER USE) A SPECIFIED PART FROM
    -- A MESSAGE

function CALCULATE_BLOCK_COUNT (MESSAGE : MSGID) return BLOCK_COUNT;
-- CALCULATES THE NUMBER OF 80 CHARACTER OUTPUT BLOCKS
    -- OCCUPIED BY A MESSAGE
-- RETURNS THE NUMBER IN STRING FORM
-- DOES NOT ALLOW FOR MCB

procedure SET_BLOCK_COUNT(MESSAGE : MSGID; COUNT : BLOCK_COUNT);
-- SET BLOCK COUNT IN MCB

task FREE_VER is
  entry FREE_VERSION(MESSAGE : MSGID);
  -- FREES HEADERS AND SEGMENTS WHICH ARE NOT IN USE BY OTHER
      -- VERSIONS
end FREE_VER;

function READ_EASN(MESSAGE : MSGID) return EXT_SERIAL_NO;
-- READ EXTENDED SERIAL NUMBER

procedure SET_CATEGORY(MESSAGE : MSGID; CATEGORY : MESSAGE_CATEGORY);
function READ_CATEGORY(MESSAGE : MSGID) return MESSAGE_CATEGORY;
-- READ AND SET MESSAGE CATEGORY (NORMAL OR SERVICE)

procedure SET_CLASS(MESSAGE : MSGID;
                    CLASS : SECURITY_CLASSIFICATION);
function READ_CLASS(MESSAGE : MSGID) return SECURITY_CLASSIFICATION;
-- SET AND READ SECURITY CLASSIFICATION OF MESSAGE

procedure SET_NUM_RIS(MESSAGE : MSGID; RI_COUNT : NUM_RIS);
```

```
function READ_NUM_RIS(MESSAGE : MSGID) return NUM_RIS;
-- SET AND READ THE NUMBER OF RIS

procedure SET_TIME_OF_RECEIPT(MESSAGE : MSGID; TOR : DATE_TIME);
function READ_TIME_OF_RECEIPT(MESSAGE : MSGID) return DATE_TIME;
-- SET AND READ TIME OF RECEIPT

procedure SET_PRECEDENCE(MESSAGE : MSGID; PREC : PRECEDENCE);
function READ_PRECEDENCE(MESSAGE : MSGID) return PRECEDENCE;
-- SET AND READ PRECEDENCE

procedure SET_LOGICAL_LINE(MESSAGE : MSGID;
                           LINE_NUMBER : LOGICAL_LINE);
function READ_LOGICAL_LINE(MESSAGE : MSGID) return LOGICAL_LINE;
-- SET AND READ LOGICAL OUTPUT LINE NUMBER

procedure SET_CHAR_TYPE(MESSAGE : MSGID; SET : CHAR_SET_TYPE);
function READ_CHAR_TYPE(MESSAGE : MSGID) return CHAR_SET_TYPE;
-- SET AND READ CHARACTER SET TYPE

procedure SET_FORMAT(MESSAGE : MSGID; FMT : FORMAT);
function READ_FORMAT(MESSAGE : MSGID) return FORMAT;
-- SET AND READ MESSAGE FORMAT

procedure FIND_CLASS(MESSAGE : MSGID;
                     CLASS : out STRING;
                     ERROR : out BOOLEAN);
procedure FIND_LMF(MESSAGE : MSGID;
                   LMF : out STRING;
                   ERROR : out BOOLEAN);
-- RETURN THE DESIRED PORTION OF THE HEADER

procedure FIND_FIRST_RI(MESSAGE : MSGID;
                        POS : out POSITION;
                        STATUS : out RI_STAT);
-- RETURNS THE POSITION OF THE FIRST RI IN A MESSAGE

procedure FIND_RI(MESSAGE : MSGID;
                  POS : in out POSITION;
                  RI : out STRING;
                  STATUS : out RI_STAT);
-- CALLED WITH POS POINTING TO AN RI
-- RETURNS THAT RI AND SETS POS TO NEXT RI

function NUM_SEGMENTS(MSG : MSGID;
                      PART : PART_NAME) return SEGMENT_NUMBER;
-- RETURNS THE NUMBER OF SEGMENTS IN A PART OF A MESSAGE

procedure GET_PART(MSG : MSGID;
                   PART : PART_NAME;
```

```
                        SEGS : out SEG_ARRAY);
    -- READS THE SEGMENTS OF A PART INTO THE ARRAY SEGS FOR USE BY HISTORY

    function GET is new GET_INTRANSIT(PART_HEADER,PART_PTR);
    function GET is new GET_INTRANSIT(VERSION_HEADER,MSGID);
    procedure FREE is new FREE_INTRANSIT(PART_HEADER,PART_PTR);
    procedure FREE is new FREE_INTRANSIT(VERSION_HEADER,MSGID);
    -- GET AND FREE STORAGE FOR VERSION_HEADERS AND PART_HEADERS
    -- IN THE INTRANSIT STORAGE AREA.  SEE MANAGE_STORAGE FOR DETAILS.

  private

    type USER_COUNTER is range 0..50;
    type PART_HEADER is
      record
        FIRST_SEGMENT : SEG_PTR;
         AST_SEGMENT  : SEG_PTR;
        SEGMENT_COUNT : SEGMENT_NUMBER;
        PART          : PART_NAME;
        USER_COUNT    : USER_COUNTER;
      end record;
    type PARTS_ARRAY is array(PART_NAME) of PART_PTR;
    type VERSION_HEADER is
      record
        CATEGORY             : MESSAGE_CATEGORY;
        CHAR_TYPE            : CHAR_SET_TYPE;
        CLASS                : SECURITY_CLASSIFICATION;
        EASN                 : EXT_SERIAL_NO;
        FMT                  : FORMAT;
        LOGICAL_OUTPUT_LINE  : LOGICAL_LINE;
        PREC                 : PRECEDENCE;
        RI_COUNT             : NUM_RIS;
        SECTIONS             : PARTS_ARRAY;
        TIME_OF_RECEIPT      : DATE_TIME;
      end record;
    type POSITION is
      record
        SEG   : SEG_PTR;
        COUNT : CHAR_COUNT;
      end record;
  end MESSAGE_OPS;
```

```
with RI_OPS, MESSAGE_OPS, GLOBAL_TYPES; use RI_OPS, MESSAGE_OPS,
    GLOBAL_TYPES;

package SERVICE_MESSAGE_OPS is
  ------------------------------------------------------------------
  -- NAME: SERVICE_MESSAGE_OPS
  -- PURPOSE: contains data definitions and operations required for
  --          generating service messages.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type SVC_MSG_TYPE is (ALL_RI_INVALID, EXCESSIVE_ROUTING_REJ,
      HI_PREC_ACC, ILLEGAL_EXCHANGE, INCORRECT_CSN,
      INPUT_SCTY_MISMATCH, INVALID_BLOCK_COUNT, INVALID_EOM_ACC,
      INVALID_EOM_REJ, INVALID_HEADER_ACC, INVALID_HEADER_REJ,
      INVALID_RI, INVALID_RI_FIELD, INVALID_SCTY_FIELD,
      INVALID_TI_ACC, INVALID_TI_REJ, NO_EOM, OPEN_CSN,
      OUTPUT_SCTY_MISMATCH, SUSPECTED_STRAGGLER,
      SUSPENDED_TRANSMISSION, TRAFFIC_CHECK, TWO_CONSEC_SOM);
  type SVC_MSG_INFO is
    record
      MSG_REF:MSGID;
      MSG_TYPE:SVC_MSG_TYPE;
      RIS:RI_PTR;
      LOW_CSN:CSN;
      HIGH_CSN:CSN;
    end record;

  task type GEN_SVC is
    entry GENERATE_SVC_MESSAGE(INFO:SVC_MSG_INFO);
  end GEN_SVC;

end SERVICE_MESSAGE_OPS;
```

```
with PHYSICAL_PORT,MANAGE_TRANSLATED_QUEUES;
use PHYSICAL_PORT,MANAGE_TRANSLATED_QUEUES;

package TASKS is
  ------------------------------------------------
  --  NAME: TASKS
  --  PURPOSE: Data structure used to correlate tasks with the line number
  --       of the physical line they are servicing.
  ------------------------------------------------

  type OUTPUT_PTR is access OUTPUT_MESSAGE;
  type SEND_II_IV_PTR is access SEND_MODE_II_IV;
  type SEND_V_PTR is access SEND_MODE_V;
  type SEND_I_PTR is access SEND_MODE_I;
  type CONTROL_PTR is access CONTROL_CHARACTER;

  type TASK_LIST is
    record
      HANDLER: MANAGER;
      OUTPUT: OUTPUT_PTR;
      SEND_II_IV: SEND_II_IV_PTR;
      SEND_V: SEND_V_PTR;
      SEND_I: SEND_I_PTR;
      GENCC,RECVCC: CONTROL_PTR;
    end record;

  TABLE: array ( PHYSICAL_LINE ) of TASK_LIST;

end TASKS;
```

```
with GLOBAL_TYPES; use GLOBAL_TYPES;

package TIME_OPS is
  ------------------------------------------------------------------
  -- NAME: TIME_OPS
  -- PURPOSE: provides operations on the type DATE_TIME
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 14, 1982
  ------------------------------------------------------------------
  function READ_CLOCK return DATE_TIME;
  -- READS THE SYSTEM TIME AND CONVERTS IT TO A JULIAN DATE_TIME

  function "<" (T1, T2 : DATE_TIME) return BOOLEAN;
  function "<=" (T1, T2 : DATE_TIME) return BOOLEAN;
  function ">" (T1, T2 : DATE_TIME) return BOOLEAN;
  function ">=" (T1, T2 : DATE_TIME) return BOOLEAN;
  -- provide comparison facilities for the Julian DATE_TIME type
end TIME_OPS;
```

## 3.2 Package Specification Dependencies

The chart in Figure 3.2-1 illustrates the package specification dependencies. This is especially useful if a package must be recompiled, since the chart shows which other packages must also be recompiled. Any package to the right of the one being recompiled that is connected by a circle must also be recompiled.

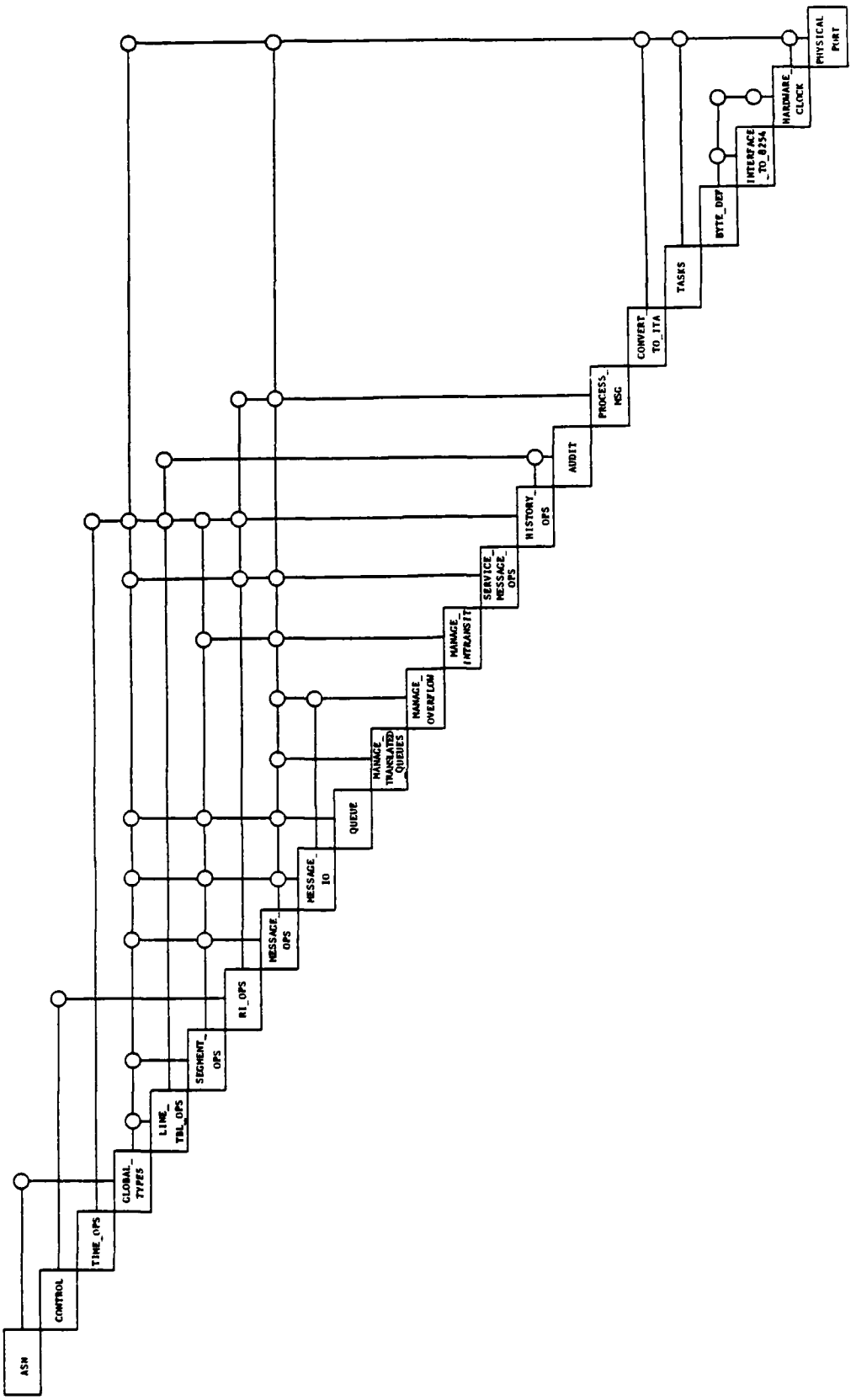



Figure 3.2-1 Package Specification Dependencies

## 3.3 Traceability Matrix

The following section shows forward and backward traceability. The forward direction is from the Requirements Specification to the Functional Decomposition Charts illustrated in Section 2.4 of this document. Reverse traceability is the opposite of the above process.

FORWARD TRACEABILITY

| REQUIREMENT | DESIGN MODULE |
|---|---|
| A11 | RUN_SWITCH |
| A121 | AUDIT.AUDIT_INTRANSIT |
| A122 | HISTORY_OPS.RE_ENTER_MESSAGES |
| A131 | MONITOR_HARDWARE_ERRORS |
| A1321 | AUDIT.AUDIT_OVERFLOW |
| A1322 | RUN_SWITCH |
| A1331 | AUDIT.AUDIT_INTERCEPT |
| A1332 | RUN_SWITCH |
| A134 | MONITOR_HARDWARE_ERRORS |
| A311 | BUILD_MESSAGE_FROM_SEGMENTS |
| A312 | BUILD_VALID_ASYNCH_SEG.RECEIVE_CHARS |
| A313 | BUILD_SYNCH_SEGMENT.RECEIVE_CHARS |
| A3141 | FORMAT_ASYNCH_SEGMENT |
| A3142 | RECOGNIZE_START_OF_MESSAGE |
| A3143 | FORMAT_ASYNCH_SEGMENT |
| A3143 | RECOGNIZE_ENDING_SEQUENCE |
| A3144 | PROCESS_CONTROL_SEQUENCE |
| A3151 | BUILD_SYNCH_SEGMENT |
| A3152 | IDENTIFY_1ST_CHARACTER |
| A3153 | VALIDATE_2ND_CHARACTER |
| A3154 | BUILD_SYNCH_SEG |
| A3154 | INTERPRET_CC |
| A3155 | VALIDATE_3RD_CHARACTER |
| A3156 | INTERPRET_CC |
| A3157 | CHECK_BLK_PARITY |
| A3161 | VALIDATE_FIRST_SEGMENT |
| A3161 | SET_ECSN |
| A3162 | CHECK_MODE_II_IV_CSN |
| A3162 | READ/SET_E/RCSN |
| A3163 | CHECK_MODE_V_CSN |
| A3163 | READ/SET_ECSN |
| A3164 | VALIDATE_FIRST_SEGMENT |
| A3164 | READ_CHNL_DES |
| A3165 | COMPLETE_TI |
| A3166 | RECOGNIZE_ENDING_SEQUENCE |
| A32 | GENERATE_CONTROL_CHARACTERS |
| A331 | VALIDATE_MESSAGE |
| A3321 | CHECK_JANAP_HEADER |
| A3321 | READ_SECURITY_PROSIGN |
| A3322 (A3343) | CHECK_RECORD_COUNT |
| A3323 (A3332,A3344,A3354) | CHECK_RIS |
| A3324 (A3334,A3345,A3354) | CHECK_EOM |
| A3325 (A3346) | CHECK_EOT |
| A3331 | CHECK_ACP_HEADER |
| A3332 | CHECK_RIS |
| A3333 | COMPLETE_ACP_HEADER |
| A3333 | READ_SECURITY_PROSIGN |
| A3334 | CHECK_EOM |
| A3341 | CHECK_LMF |
| A3342 | CHECK_HPJ_HEADER |

## FORWARD TRACEABILITY

| REQUIREMENT | DESIGN MODULE |
|---|---|
| A3342 | READ_SECURITY_PROSIGN |
| A3343 | CHECK_RECORD_COUNT |
| A3344 | CHECK_RIS |
| A3345 | CHECK_EOM |
| A3346 | CHECK_EOT |
| A3351 | CHECK_HPA_HEADER |
| A3352 | CHECK_RIS |
| A3353 | COMPLETE_HPA_HEADER |
| A3353 | READ_SECURITY_PROSIGN |
| A3354 | CHECK_EOM |
| A341 | PROCESS_MCB |
| A342 | BUILD_JANAP_MCB |
| A343 | BUILD_ACP_MCB |
| A344 | MODIFY_MCB |
| A345 | COMPLETE_MCB |
| A35 | GENERATE_SERVICE_MESSAGE |
| A361 | STORE_REFERENCE |
| A361 | PROCESS_MCB |
| A361 | MANAGE_OVERFLOW.MOVE_TO_OVERFLOW |
| A361 | QUEUE_NORM_MSG_BY_PRECEDENCE |
| A361 | QUEUE_SVS_MSG_BY_PRECEDENCE |
| A361 | HISTORY_OPS.ENTER_HIST.WRITE |
| A362 | MANAGE_INTRANSIT.CALCULATE_THRESHOLD |
| A363 | MANAGE_OVERFLOW.RECOVER_OVERFLOW |
| A363 | MANAGE_MESSAGE_QUEUES.QUEUE_REINTROD |
| A363 | HISTORY_OPS.ENTER_HIST.WRITE_HISTORY |
| A364 | RETRV_NEXT_MSG_FOR_PROCESSING |
| A364 | QUQUE_FROM_FRONT |
| A411 | GET_RIS |
| A412 | GET_RIS |
| A413 | SEGREGATE_ROUTING_LINE |
| A413 | GET_RIS |
| A421 | TRANSLATE_MESSAGE |
| A4221 | TRANSLATE_MESSAGE |
| A4222 | CONVERT_TO_JANAP |
| A4223 | CONVERT_TO_ACP |
| A4224 | TRANSLATE_MESSAGE |
| A4225 | REMOVE_DBLS |
| A4231 | TRANSLATE_MESSAGE |
| A4232 | TRANSLATE_TO_ASCII |
| A4233 | TRANSLATE_TO_ITA_2 |
| A4234 | TRANSLATE_MESSAGE |
| A4235 | TRANSLATE_TO_CARD |
| A4236 | TRANSLATE_FROM_CARD |
| A4237 | TRANSLATE_MESSAGE |
| A4238 | FILL_LAST_BLOCK |
| A431 | MANAGE_TRANSLATED_MESSAGE_QUEUES |
| A432 | MOVE_FROM_INTERCEPT_TO_INTRANSIT |
| A433 | RETRIEVE_NEXT_MESSAGE |
| A433 | QUEUE_FROM_FRONT |

FOR *RD TRACEABILITY

| REQUIREMENT | DESIGN MODULE |
|---|---|
| A511 | MESSAGE_OPS.READ_CONTINUOUS |
| A512 | VALIDATE_RI |
| A513 | VALIDATE_LMF |
| A513 | FIND_LMF |
| A514 | VALIDATE_SECURITY |
| A514 | READ_LOGICAL_LINE |
| A514 | FIND_CLASS |
| A515 | SEND_ASYNCH |
| A531 | GEN_FRAME_CHARS |
| A5321 | FRAME_BLOCK |
| A5322 | FRAME_BLOCK |
| A5323 | FRAME_BLOCK |
| A533 | OUTPUT_MESSAGE |
| A541 | SEND_SYNCH |
| A5411 | SEND_SYNCH |
| A5411 | TRANSMIT_SYNCH_SYNCH |
| A5412 | SEND_SYNCH |
| A5412 | TRANSMIT_SYNCH_CHARS |
| A5413 | SEND_SYNCH |
| A5413 | TRANSMIT_SYNCH_CHARS |
| A5414 | SEND_SYNCH |
| A5414 | TRANSMIT_SYNCH_CHARS |
| A5421 | GENERATED_CONTROL_CHARS |
| A5421 | RECEIVED_CONTROL_CHARS |
| A5422 | GENERATED_CONTROL_CHARS |
| A5422 | RECEIVED_CONTROL_CHARS |
| A5423 | GENERATED_CONTROL_CHARS |
| A5423 | RECEIVED_CONTROL_CHARS |
| A5424 | GENERATED_CONTROL_CHARS |
| A5424 | RECIEVED_CONTROL_CHARS |
| A5425 | GENERATED_CONTROL_CHARS |
| A5425 | RECEIVED_CONTROL_CHARS |
| A5426 | GENERATED_CONTROL_CHARS |
| A5426 | RECEIVED_CONTROL_CHARS |
| A543 | OUTPUT_MES AGE |
| A543 | READ_CHANNEL_MODE |
| A551 | HISTORY_OPS.ENTER_HIST.WRITE |
| A552 | HISTORY_OPS.ENTER_HIST.WRITE |
| A553 | HISTORY_OPS.ENTER_HIST.WRITE |
| A554 | HISTORY_OPS.ENTER_HIST.WRITE |
| A555 | HISTORY_OPS.ENTER_HIST.WRITE |
| A556 | HISTORY_OPS.ENTER_HIST.WRITE |
| NON FUNCTIONAL REQUIREMENTS | |
| I.SENTENCE 1 | RECEIVE_VALID_MESSAGE |
| I.SENTENCE 1 | MANAGE_MESSAGE_QUEUES |
| I.SENTENCE 2 | THROTTLE_INPUT |
| I.A & B | RECEIVE_CHARS |
| II.A.(1) | RECEIVE_CHARS |
| II.A.(2) | EVERYTHING |

## FORWARD TRACEABILITY

| REQUIREMENT | DESIGN MODULE |
|---|---|
| II.A.(3) | RECEIVE_CHARS |
| II.A.(4) | RECEIVE_CHARS |
| II.A.(5) | BUILD_VALID_SYNCH_SEG |
| II.A.(6) | BUILD_VALID_SYNCH_SEG |
| II.A.(7) | FORMAT_SYNCH_SEG |
| II.A.(8) | FORMAT_SYNCH_SEG.RECEIVE_CHAR |
| II.A.(9) | PROCESS_MCB |
| II.A.(10) | RECEIVE_CHAR |
| II.A.(11) | FORMAT_SYNCH_SEG.RECEIVE_CHAR |
| II.A.(12) | FORMAT_SYNCH_SEG.RECEIVE_CHAR |
| II.B.(1) | FORMAT_SYNCH_SEG.RECEIVE_CHAR |
| II.B.(2) | SAME AS II.A.(1) - II.A.(11) |
| III.A.(1) & (2) | BUILD_VALID_ASYNCH_SEG.RECEIVE_CHAR |
| III.A.(1) & (2) | XMIT_ASYNCH_CHAR |
| III.A.(3) | THE WHOLE SWITCH |
| III.B.(1) & (2) & (3) | BUILD_VALID_ASYNCH_SEG.RECEIVE_CHAR |
| III.B.(1) & (2) & (3) | XMIT_ASYNCH_CHAR |
| IV.A.(1) & (2) | RECEIVE_CHAR AND XMIT_CHAR |
| IV.B.(1) & (2) | RECEIVE_CHAR AND XMIT_CHAR |
| V.SENTENCE 1 | SEGMENT_OPS |
| V.SENTENCE 2 | NOT APPLICABLE |
| V.SENTENCE 3 | PACKAGE MANAGE_INTRANSIT |
| V.SENTENCE 4 | NOT APPLICABLE |
| V.SENTENCE 5 | NOT APPLICABLE |
| V.SENTENCE 6 (DISTRIBUTION) | NOT APPLICABLE |
| VI. | UNDETERMINABLE AT THIS TIME |
| VII. | UNDETERMINABLE AT THIS TIME |
| IX. | RI_OPS |
| X. | MANAGE_INTRANSIT |
| XI. | UNDETERMINABLE AT THIS TIME |

| CONCURRENCY CHART | DESIGN DOCUMENT COMPONENT |
|---|---|
| STARTUP/RECOVERY | RUN SWITCH |
| ASSEMBLE MESSAGES | RECEIVE VALID MESSAGE |
| PROCESS MESSAGES | PROCESS MESSAGE |
| TRANSMIT MESSAGES | OUTPUT MESSAGE |
| OPERATOR INTERFACE | MONITOR SWITCH |

## BACKWARD TRACEABILITY

D.I.= DESIGN IMPLEMENTATION

| | |
|---|---|
| ADD_TEXT | D.I.- A3143, A3152, A3153, A3154 |
| ASN.GET | Fulfills requirement that unique ASN is assigned to each incoming message. |
| ASYNCH_SEG.RECOGNIZE_END_OF_PART | D.I.-breaks the transmission identifier of asynch messages into segment. |
| AUDIT.AUDIT_INTERCEPT | A1331 |
| AUDIT.AUDIT_INTRANSIT | A121 |
| AUDIT.AUDIT_OVERFLOW | A1321 |
| BUILD_ACP_MCB | A343 |
| BUILD_JANAP_MCB | A342 |
| BUILD_MESSAGE_FROM_SEGMENTS | A311 |
| BUILD_SYNCH_SEG | A3154 |
| BUILD_SYNCH_SEGMENT | A3151 |
| BUILD_SYNCH_SEGMENT.RECEIVE_CHARS | A313 |
| BUILD_VALID_ASYNCH_SEG.RECEIVE_CHAR | III.B.(1) & (2) & (3) |
| BUILD_VALID_ASYNCH_SEG.RECEIVE_CHAR | III.A.(1) & (2) |
| BUILD_VALID_ASYNCH_SEG.RECEIVE_CHARS | A312 |
| BUILD_VALID_SYNCH_SEG | II.A.(5) |
| BUILD_VALID_SYNCH_SEG | II.A.(6) |
| CHECK_ACP_HEADER | A3331 |
| CHECK_BLK_PARITY | A3157 |
| CHECK_EOM | A3324,A3334,A3345,A3354 |
| CHECK_EOT | A3325,A3346 |
| CHECK_HPA_HEADER | A3351 |
| CHECK_HPJ_HEADER | A3342 |
| CHECK_ICD | A3164 |
| CHECK_JANAP_HEADER | A3321 |
| CHECK_LMF | A3341 |
| CHECK_MODE_II_IV_CSN | A3162 |
| CHECK_MODE_V_CSN | A3163 |
| CHECK_RECORD_COUNT | A3322,A3343 |
| CHECK_RIS | A3352 |
| CHECK_RIS | A3323,A3332,A3344,A3354 |
| COMPLETE_ACP_HEADER | A3333 |
| COMPLETE_HPA_HEADER | A3353 |
| COMPLETE_MCB | A345 |
| COMPLETE_TI | A3165 |
| CONVERT_TO_ACP | A4223 |
| CONVERT_TO_JANAP | A4222 |
| DEQUEUE_MSG_FOR_OVERFLOW | Implicit requirement- lower precedence messages overflow. |
| FILL_LAST_BLOCK | A4238 |
| FIND_CLASS | A514 |
| FIND_LMF | A513 |
| FORMAT_ASYNCH_SEGMENT | A3143 |
| FORMAT_ASYNCH_SEGMENT | A3141 |
| FORMAT_SYNCH_SEG | II.A.(7) |

## BACKWARD TRACEABILITY

| DESIGN MODULE | REQUIREMENT |
|---|---|
| FORMAT_SYNCH_SEG.RECEIVE_CHAR | II.A.(8) |
| FORMAT_SYNCH_SEG.RECEIVE_CHAR | II.A.(11) |
| FORMAT_SYNCH_SEG.RECEIVE_CHAR | II.A.(12) |
| FORMAT_SYNCH_SEG.RECEIVE_CHAR | II.B.(1) |
| FRAME_BLOCK | A5323 |
| FRAME_BLOCK | A5321 |
| FRAME_BLOCK | A5322 |
| GENERATED_CONTROL_CHARS | A5425 |
| GENERATED_CONTROL_CHARS | A5421 |
| GENERATED_CONTROL_CHARS | A5422 |
| GENERATED_CONTROL_CHARS | A5426 |
| GENERATED_CONTROL_CHARS | A5423 |
| GENERATED_CONTROL_CHARS | A5424 |
| GENERATE_CONTROL_CHARACTERS | A32 |
| GENERATE_SERVICE_MESSAGE | A35 |
| GEN_FRAME_CHARS | A531 |
| GET_RIS | A413 |
| GET_RIS | A411 |
| GET_RIS | A412 |
| GET_SEG | D.I.- break messages into segments. |
| HISTORY_OPS.ENTER_HIST.WRITE | A361 |
| HISTORY_OPS.ENTER_HIST.WRITE | A555 |
| HISTORY_OPS.ENTER_HIST.WRITE | A551 |
| HISTORY_OPS.ENTER_HIST.WRITE | A554 |
| HISTORY_OPS.ENTER_HIST.WRITE | A552 |
| HISTORY_OPS.ENTER_HIST.WRITE | A553 |
| HISTORY_OPS.ENTER_HIST.WRITE | A556 |
| HISTORY_OPS.ENTER_HIST.WRITE_HISTORY | A363 |
| HISTORY_OPS.RE_ENTER_MESSAGES | A122 |
| IDENTIFY_1ST_CHARACTER | A3152 |
| INTERPRET_CC | A3154 |
| INTERPRET_CC | A3156 |
| MANAGE_INTRANSIT | X. |
| MANAGE_INTRANSIT.CALCULATE_THRESHOLD | A362 |
| MANAGE_MESSAGE_QUEUES | I.SENTENCE 1 |
| MANAGE_MESSAGE_QUEUES.QUEUE_REINTROD | A363 |
| MANAGE_OVERFLOW.MOVE_TO_OVERFLOW | A361 |
| MANAGE_OVERFLOW.RECOVER_OVERFLOW | A363 |
| MANAGE_TRANSLATED_MESSAGE_QUEUES | A431 |
| MANAGE_TRANSLATED_MESSAGE_QUEUES | A433 |
| MESSAGE_IO | Reads/writes to hardware file (tape, disk, virtual memory). |
| MESSAGE_OPS.ATTACH_SEGMENT | D.I.-of internal storage of messages |
| MESSAGE_OPS.CALCULATE_BLOCK_COUNT | D.I.-of message block counts |
| MESSAGE_OPS.ESTABLISH_VERSION | D.I.-of internal storage of message |
| MESSAGE_OPS.FREE_PART | D.I.-of internal storage of messages |
| MESSAGE_OPS.READ_CONTINUOUS | A511 |
| MESSAGE_OPS.SET_VLOCK_COUNT | Req. to keep block count in MCR update |
| MESSAGE_OPS_OPEN_FOR_READ | D.I.-of internal storage of messages |
| MESSAGE_OPS_READ_CONTINUOUS | D.I.-of internal storage of messages |

## BACKWARD TRACEABILITY

| DESIGN MODULE | REQUIREMENT |
|---|---|
| MODIFY_MCB | A344 |
| MONITOR HARDWARE ERRORS | A134 |
| MONITOR HARDWARE ERRORS | A131 |
| MOVE_FROM_INTERCEPT_TO_INTRANSIT | A432 |
| MOVE_TO_INTERCEPT | Partially fulfills A431 |
| MSG_OPS.FREE_VERSION | D.I.- removes outdated messages |
| MSG_OPS.READ_CATEGORY | Part of requirement of A364 |
| MSG_OPS.READ_PRECEDENCE | Part of requirement of A364 and A433 |
| MSG_OPS.READ_TIME_OF_RECEIPT | Part of requirement of A364 and A433 |
| MSG_OPS.SET_CATEGORY | Part of requirement of A364 |
| MSG_OPS.SET_PRECEDENCE | Part of requirement of A364 and A433 |
| MSG_OPS.SET_TIME_OF_RECEIPT | Part of requirement of A364 and A433 |
| NOTIFY_OPERATOR | Implementation of 'Notify Operator' |
| OUTPUT_MESSAGE | A543 |
| OUTPUT_MESSAGE | A533 |
| PACKAGE MANAGE_INTRANSIT | V.SENTENCE 3 |
| PROCESS_CONTROL_SEQUENCE | A3144 |
| PROCESS_MCB | A341 |
| PROCESS_MCB | II.A.(9) |
| PROCESS_MCB | A361 |
| PROCESS_MESSAGE | Control of A4 |
| QUEUE_FROM_BACK | Implicit requirement |
| QUEUE_MSG_BY_PRECEDENCE_AND_BY_TOR | Partially fulfills A431 |
| QUEUE_NORM_MSG_BY_PRECEDENCE | A361 |
| QUEUE_ON_FRONT | Implicit requirement |
| QUEUE_ON_F_BACK | A361 |
| QUEUE_SVS_MSG_BY_PRECEDENCE | A361 |
| READ_CATEGORY | D.I. to accomplish multiple routing |
| READ_CHANNEL_DES | A3164 |
| READ_CHANNEL_MODE | D.I. to make decision of A543 |
| READ_CHAR_SET | D.I. to accomplish multiple routing |
| READ_CLASS | D.I. to accomplish multiple routing |
| READ_EASN | D.I. to accomplish multiple routing |
| READ_ECSN | A3162, A3163 |
| READ_FORMAT | D.I. to accomplish multiple routing |
| READ_LOGICAL_LINE | A514 |
| READ_RCSN | A3163 |
| READ_SECURITY_PROSIGN | A3321,A3333,A3342,A3353 |
| READ_TOR | D.I. to accomplish multiple routing |
| READY_LINE_NO | D.I.- sets line status to 'ready' for next message transmission. |
| RECEIVED_CONTROL_CHARS | A5426 |
| RECEIVED_CONTROL_CHARS | A5425 |
| RECEIVED_CONTROL_CHARS | A5421 |
| RECEIVED_CONTROL_CHARS | A5422 |
| RECEIVED_CONTROL_CHARS | A5423 |
| RECEIVE_CHAR | II.A.(10) |
| RECEIVE_CHAR AND XMIT_CHAR | IV.B.(1) & (2) |
| RECEIVE_CHAR AND XMIT_CHAR | IV.A.(1) & (2) |
| RECEIVE_CHARS | II.A.(3) |

## BACKWARD TRACEABILITY

| DESIGN MODULE | REQUIREMENT |
|---|---|
| RECEIVE_CHARS | II.A.(1) |
| RECEIVE_CHARS | II.A.(4) |
| RECEIVE_CHARS | I.A & B |
| RECEIVE_VALID_MESSAGE | D.I.- removes outdated messages |
| RECEIVE_VALID_MESSAGE | I.SENTENCE 1 |
| RECIEVED_CONTROL_CHARS | A5424 |
| RECOGNIZE_ENDING_SEQUENCE | A3166 |
| RECOGNIZE_ENDING_SEQUENCE | A3143 |
| RECOGNIZE_START_OF_MESSAGE | A3142 |
| REFLECT_STATUS | Part of control of A3 |
| REMOVE_DBLS | A4225 |
| RESET_SEG | D.I. |
| RETRIEVE_NEXT_MESSAGE | Partially fulfills A433 |
| RETRV_NEXT_MSG_FOR_PROCESSING | A364 |
| RI_OPS | IX. |
| RI_OPS.FIND_FIRST_RI | D.I. |
| RI_OPS.FIND_RI | Same as FIND_FIRST_RI |
| RI_OPS.READ_RI_TABLE | D.I. |
| ROUTE_MESSAGE | Control of A41 |
| RUN_SWITCH | A1322 |
| RUN_SWITCH | A11 |
| RUN_SWITCH | A1332 |
| SCAN_FOR_COLLECTIVE_RIs | Partial implementation of A345 |
| SEGMENT_OPS | V.SENTENCE 1 |
| SEGMENT_OPS.NEXT_SEGMENT | D.I. of internal storage of messages |
| SEGMENT_OPS.PRIOR_SEGMENT | D.I. of internal storage of messages |
| SEGMENT_OPS.READ_CHARACTER_COUNT | D.I. |
| SEGMENT_OPS.READ_CHARS | D.I. of internal storage of segments |
| SEGMENT_OPS.READ_PART | D.I. |
| SEGMENT_OPS.READ_SEGMENT_NUMBER | Implicit requirement |
| SEGMENT_OPS.READ_TIME | See TIME_OPS.READ_CLOCK |
| SEGMENT_OPS.SET_PART | D.I. |
| SEGMENT_OPS.SET_SEGMENT_NUMBER | Implicit requirement |
| SEGMENT_OPS.SET_TIME | See TIME_OPS.READ_CLOCK |
| SEGMENT_OPS.WRITE_SPECIFIC | D.I. of internal storage of segments |
| SEGMENT_OPS.WRITE_TO_PART | D.I. of internal storage of messages |
| SEGREGATE_ROUTING_LINE | A413 |
| SEND_ASYNCH | A515 |
| SEND_SYNCH | A5412 |
| SEND_SYNCH | A541 |
| SEND_SYNCH | A5414 |
| SEND_SYNCH | A5411 |
| SEND_SYNCH | A5413 |
| SET_CATEGORY | D.I. to accomplish multiple routing |
| SET_CHAR_SET | D.I. to accomplish multiple routing |
| SET_CLASS | D.I. to accomplish multiple routing |
| SET_ECSN | A3161, A3162, A3163 |
| SET_FORMAT | D.I. to accomplish multiple routing |
| SET_RCSN | A3163 |
| SET_TOR | D.I. to accomplish multiple routing |

## BACKWARD TRACEABILITY

| DESIGN MODULE | REQUIREMENT |
|---|---|
| STORE_REFERENCE | A361 |
| THROTTLE_INPUT | I.SENTENCE 2 |
| TIME_OPS._READ_CLOCK | Implicit requirement |
| TRANSLATE_FROM_CARD | A4236 |
| TRANSLATE_MESSAGE | A4237 |
| TRANSLATE_MESSAGE | A4221 |
| TRANSLATE_MESSAGE | A4234 |
| TRANSLATE_MESSAGE | A421 |
| TRANSLATE_MESSAGE | A4224 |
| TRANSLATE_MESSAGE | A4231 |
| TRANSLATE_TO_ASCII | A4232 |
| TRANSLATE_TO_CARD | A4235 |
| TRANSLATE_TO_ITA_2 | A4233 |
| TRANSMIT_SYNCH_CHARS | A5412 |
| ..ANSMIT_SYNCH_CHARS | A5414 |
| TRANSMIT_SYNCH_CHARS | A5413 |
| TRANSMIT_SYNCH_SYNCH | A5411 |
| VALIDATE_2ND_CHARACTER | A3153 |
| VALIDATE_83RD_CHARACTER | A3155 |
| VALIDATE_ACP_MSG | Control of A333 |
| VALIDATE_FIRST_SEGMENT | A161 |
| VALIDATE_HPA_MSG | Control of A335 |
| VALIDATE_HPJ_MSG | Control of A334 |
| VALIDATE_JANAP_MESSAGE | A332 |
| VALIDATE_JANAP_MSG | Control of A332 |
| VALIDATE_LMF | A513 |
| VALIDATE_MESSAGE | A331 |
| VALIDATE_MESSAGE_HEADER | Control of A51 |
| VALIDATE_RI | A512 |
| VALIDATE_SECURITY | A514 |
| XMIT_ASYNCH_CHAR | III.B.(1) & (2) & (3) |
| XMIT_ASYNCH_CHAR | III.A.(1) & (2) |

## 3.4 Data Structures

The following section illustrates the detailed structure of the site dependent database tables.

### 3.4.1 Line Table Operations

The table in this section is used by the message switch software to determine how to initialize the serial data channel hardware and the proper protocol to handle the incoming and outgoing messages.

```
with GLOBAL_TYPES; use GLOBAL_TYPES;

package LINE_TBL_OPS is
  ------------------------------------------------
  --  NAME: LINE_TBL_OPS
  --  PURPOSE: Data structures that contain the information describing
  --       the characteristics of each line, and the subprograms needed to
  --       access the information. ( Note: All accesses must use CONTROL
  --       as these data structrues are shared. )
  ------------------------------------------------
  type TRANSMISSION_MODE is ( SYNCH_BLK_BY_BLK, SYNCH_CONTINUOUS,
      ASYNCH_NORMAL, ASYNCH_STEPPED );
  type FORMAT_TYPE is ( ANY, JANAP, ACP, ACP_MOD );
  type LEVEL_TYPE is range 5..8;
  VALID_LEVEL : array( 1..2 ) of LEVEL_TYPE := ( 5, 8 );
  type PHYSICAL_LINE is range 0..50;
  type LOOP_SPEED is delta 0.1 range 45..9600;
  VALID_SPEED : array( 1..12 ) of LOOP_SPEED := ( 45, 50, 56.9, 74.2,
      75, 150, 300, 600, 1200, 2400, 4800, 9600 );
  type COMMUNITIES_SERVED is ( R, U, RU, Y, RY, UY, RUY );
  type FIRST_LINK is new BOOLEAN;
  MAX_NO_RIS_PER_DELIVERY : array ( 0..4 ) of INTEGER := ( 50,
      500, 1, 6, 14 );
  type XTS_TYPE is new BOOLEAN;
  type SOM_SEQ_TYPE is ( FULL, ABBREV );
  subtype LMF is STRING( 1..2 );
  subtype MEDIA is STRING( 1..1 );
  type MEDIAS is ( 'A', 'T', 'C', 'Z' );
  type DIRECTION_TYPE is ( BOTH, INPUT, OUTPUT );
  type NO_STOP_BITS is range 1..2;
  type SPEC_TERM_TYPE is ( ACCES, INTERSWITCH, TECHNICAL_CONTROL );
  type PHYS_IDS;
  type PHYS_PTR is access PHYS_IDS;
  type PHYS_IDS is
    record
      PHYS_LINE:PHYSICAL_LINE;
      NEXT:PHYS_PTR := null;
    end record;
  type LOGICAL_IDS;
  type LOGICAL_PTR is access LOGICAL_IDS;
  type LOGICAL_IDS is
    record
      LOG_LINE: LOGICAL_LINE;
      NEXT: LOGICAL_PTR := null;
    end record;

  procedure APPEND_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
      PHYSICAL_LINE );
    --  Add a physical line to the normal list of lines for a logical
    --       line.
```

```
procedure DROP_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
    PHYSICAL_LINE );
  -- Drop a physical line from the normal list of lines for a logical
  --      line.
procedure APPEND_ALT( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
    PHYSICAL_LINE );
  -- Add a physical line to the alternate list of lines for a logical
  --      line.
procedure DROP_ALT ( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
    PHYSICAL_LINE );
  -- Drop a physical line from the alternate list of lines for a
  --      logical line.
procedure BEGIN_ALT( LOG_LINE: LOGICAL_LINE );
  -- Begin using the alternate list of lines.
procedure END_ALT( LOG_LINE: LOGICAL_LINE );
  -- Begin using the normal list of lines.
function IS_ALT( LOG_LINE: LOGICAL_LINE ) return BOOLEAN;
  -- Determine if the alternate list of lines is being used.
function VALID_PHYS_LINE( LOG_LINE: LOGICAL_LINE; PHYS_LINE:
    PHYSICAL_LINE ) return BOOLEAN;
  -- Determine if this physical line is in the list of lines
  --      being used by this logical line.
procedure MAKE_AVAILABLE( PHYS_LINE: PHYSICAL_LINE );
  -- Mark this line as available for use.
procedure NOT_AVAILABLE( PHYS_LINE: PHYSICAL_LINE );
  -- Mark this line as not available for use.
function IS_AVAILABLE( LOG_LINE: LOGICAL_LINE ) return BOOLEAN;
  -- Determine if this line is available for use.
procedure SET_LOGICAL( LOG_LINE: LOGICAL_LINE; FORMAT: FORMAT_TYPE;
    PREFERRED_LMF: LMF; LEVEL: LEVEL_TYPE;
    COMMUNITY: COMMUNITIES_SERVED; XTS: XTS_TYPE; MAX_RIS: INTEGER;
    CHARACTER_SET: CHAR_SET_TYPE; SPEC_TERM: SPEC_TERM_TYPE );
  -- Set the descriptive characteristics of a logical line
function READ_FORMAT( LOG_LINE: LOGICAL_LINE ) return FORMAT_TYPE;
  -- Provide the format for this logical line.
function READ_PREFERRED_LMF( LOG_LINE: LOGICAL_LINE ) return LMF;
  -- Provide the preferred line media format for this logical line.
function READ_LEVEL( LOG_LINE: LOGICAL_LINE ) return LEVEL_TYPE;
  -- Provide the encoding level for this logical line.
function READ_COMMUNITY( LOG_LINE: LOGICAL_LINE ) return
    COMMUNITIES_SERVED;
  -- Provide the communities served by this logical line.
function READ_XTS( LOG_LINE: LOGICAL_LINE ) return XTS_TYPE;
  -- Provide the XTS of this logical line.
function READ_MAX_RIS( LOG_LINE: LOGICAL_LINE ) return INTEGER;
  -- Provide maximum number of routing indicators permitted in a
  --      message on this logical line.
function READ_CHARACTER_SET( LOG_LINE: LOGICAL_LINE ) return
    CHAR_SET_TYPE;
  -- Provide the character set used on this logical line.
```

```
function READ_SPEC_TERM( LOG_LINE: LOGICAL_LINE ) return
    SPEC_TERM_TYPE;
  -- Provide the SPEC_TERM of this logical line.
procedure SET_PHYSICAL( PHYS_LINE: PHYSICAL_LINE; CHNL_MODE:
    CHANNEL_MODE; XMIT_MODE: TRANSMISSION_MODE; DIRECTION:
    DIRECTION_TYPE; SECURITY: SECURITY_CLASSIFICATION; DESIGNATOR:
    CHANNEL_DES; SOM_SEQ: SOM_SEQ_TYPE; STOP_BITS:
    NO_STOP_BITS );
  -- Set the descriptive characteristics of this physical line.
procedure SET_OCSN( PHYS_LINE: PHYSICAL_LINE; OCSN: CSN );
  -- Set the OCSN for this physical line.
procedure INCREMENT_OCSN( PHYS_LINE: PHYSICAL_LINE );
  -- Increment the OCSN for this physical line.
procedure SET_ECSN( PHYS_LINE: PHYSICAL_LINE; ECSN: CSN );
  -- Set the ECSN for this physical line.
procedure SET_RCSN( PHYS_LINE: PHYSICAL_LINE; RCSN: CSN );
  -- Set the RCSN for this physical line.
procedure INCREMENT_OCSN( PHYS_LINE: PHYSICAL_LINE; OCSN: CSN );
  -- Increment the OCSN for this physical line.
procedure INCREMENT_ECSN( PHYS_LINE: PHYSICAL_LINE; ECSN: CSN );
  -- Increment the ECSN for this physical line.
procedure INCREMENT_RCSN( PHYS_LINE: PHYSICAL_LINE; RCSN: CSN );
  -- Increment the RCSN for this physical line.
function READ_CHANNEL_MODE( PHYS_LINE: PHYSICAL_LINE ) return
    CHANNEL_MODE;
  -- Provide the channel mode of this physical line.
function READ_SECURITY_CLASSIFICATION( PHYS_LINE: PHYSICAL_LINE ) return
    SECURITY_CLASSIFICATION;
  -- Provide security classification of this physical line.
function READ_TRANSMISSION_MODE( PHYS_LINE: PHYSICAL_LINE ) return
    TRANSMISSION_MODE;
  -- Provide transmission mode of this physical line.
function READ_DIRECTION( PHYS_LINE: PHYSICAL_LINE ) return
    DIRECTION_TYPE;
  -- Provide transmission direction of this physical line.
function READ_DESIGNATOR( PHYS_LINE: PHYSICAL_LINE ) return
    CHANNEL_DES;
  -- Provide channel designator of this physical line.
function READ_SOM_SEQ( PHYS_LINE: PHYSICAL_LINE ) return SOM_SEQ_TYPE;
  -- Provide the start of message sequence for this physical line.
function READ_STOP_BITS( PHYS_LINE: PHYSICAL_LINE ) return NO_STOP_BITS;
  -- Provide the number of stop bits used on this physical line.
function READ_OCSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
  -- Provide current value of OCSN for this physical line.
function READ_ECSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
  -- Provide current value of ECSN for this physical line.
function READ_RCSN( PHYS_LINE: PHYSICAL_LINE ) return CSN;
  -- Provide current value of RCSN for this physical line.
procedure SET_CURRENT_PRECEDENCE( PHYS_LINE: PHYSICAL_LINE;
    CURRENT_PRECEDENCE: PRECEDENCE );
```

```
    --  Record the precedence of the message passed to this physical
    --      line for transmission.
    function READ_CURRENT_PRECEDENCE( PHYS_LINE: PHYSICAL_LINE )
      return PRECEDENCE;
    --  Provide current precedence for this physical line.
    procedure SET_START_TIME( PHYS_LINE: PHYSICAL_LINE );
    --  Record time that a message was passed to this physical line
    --      for transmission.
    function READ_START_TIME( PHYS_LINE: PHYSICAL_LINE ) return
      DATE_TIME;
    --  Provide time that this physical line was last passed a
    --      message to transmit.
    procedure SET_INTERCEPT( LOG_LINE: LOGICAL_LINE;
      INTERCEPT_PRECEDENCE: PRECEDENCE; INTERCEPT_MEDIA: MEDIA );
    --  Set intercept flags for specified criteria.
    procedure RESET_INTERCEPT( LOG_LINE: LOGICAL_LINE;
      INTERCEPT_PRECEDENCE: PRECEDENCE; INTERCEPT_MEDIA: MEDIA );
    --  Reset intercept flags for specified criteria.
    function IS_INTERCEPT( LOG_LINE: LOGICAL_LINE;
      THIS_PRECEDENCE: PRECEDENCE; THIS_MEDIA: MEDIA ) return BOOLEAN;
    --  Determine if specified criteria messages are to be intercepted.

    procedure SET_KILL( PHYS_LINE: PHYSICAL_LINE );
    --  Set kill flag for this physical line.
    procedure RESET_KILL( PHYS_LINE: PHYSICAL_LINE );
    --  Reset kill flag for this physical line.
    function IS_KILL( PHYS_LINE: PHYSICAL_LINE ) return BOOLEAN;
    --  Determine if kill flag is set for this physical line.
    procedure SET_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE );
    --  Set kill when queue empty flag for this physical line.
    procedure RESET_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE );
    --  Reset kill when queue empty flag for this physical line.
    function IS_KILL_ON_EMPTY( PHYS_LINE: PHYSICAL_LINE ) return BOOLEAN;
    --  Determine if kill when queue empty flag is set for this physical
    --      line.

private
  type LOGICAL_PORT is
    record
      INTERCEPT: array ( PRECEDENCE, MEDIAS ) of BOOLEAN;
      NAMESAKES: LOGICAL_IDS;
      FORMAT: FORMAT_TYPE := ANY;
      PREFERRED_LMF: LMF;
      LEVEL: LEVEL_TYPE := VALID_LEVEL( 2 );
      ALT_PORTS, PHYS_PORTS: PHYS_PTR := null;
      COMMUNITIES: COMMUNITIES_SERVED := R;
      XTS: XTS_TYPE := FALSE;
      MAX_RIS: INTEGER := MAX_NO_RIS_PER_DELIVERY( 0 );
      CHARACTER_SET: CHAR_SET_TYPE := ANY;
      SPEC_TERM: SPEC_TERM_TYPE := ACCES;
```

```
        AVAILABLE: BOOLEAN := FALSE;
        ALT_ROUTING: BOOLEAN := FALSE;
      end record;
    LOGICAL_TABLE : array ( LOGICAL_LINE'FIRST .. LOGICAL_LINE'LAST ) of
        LOGICAL_PORT;
    type PHYSICAL_PORT( XMIT_MODE: TRANSMISSION_MODE :=
        SYNCH_BLK_BY_BLK ) is
      record
        CHNL_MODE: CHANNEL_MODE;
        CURRENT_PRECEDENCE: PRECEDENCE;
        START_TIME: DATE_TIME;
        DIRECTION: DIRECTION_TYPE := BOTH;
        SECURITY: SECURITY_CLASSIFICATION;
        SPEED: LOOP_SPEED;
        KILL, KILL_ON_EMPTY,
        AVAILABLE: BOOLEAN := FALSE;
        DESIGNATOR: CHANNEL_DES;
        case XMIT_MODE is
          when ASYNCH_NORMAL | ASYNCH_STEPPED =>
            STOP_BITS: NO_STOP_BITS := 1;
            SOM_SEQ: SOM_SEQ_TYPE := FULL;
            ECSN, OCSN, RCSN: CSN := 0;
          when others =>
            null;
        end case;
      end record;
    PHYSICAL_TABLE : array ( PHYSICAL_LINE'FIRST .. PHYSICAL_LINE'LAST )
        of PHYSICAL_PORT;

end LINE_TBL_OPS;
```

### 3.4.2 Routing Indicator Operations

The table in the following section is used by the message processing software to determine how many copies must be made and which logical channels to route them over. These tables are initialized by the switch operator at installation time and may be changed from the operator console by the "Run Switch" software.

```
with GLOBAL_TYPES; use GLOBAL_TYPES;

package RI_OPS is
  ------------------------------------------------------------------
  -- NAME: RI_OPS
  -- PURPOSE: contains data definitions and procedures required to
  --          decode routing indicators and determine the required
  --          routing.
  -- PROGRAMMER: Paul Dobbs
  -- DATE: May 17, 1982
  ------------------------------------------------------------------

  type PORT_ENTRY;
  type ENTRY_PTR is access PORT_ENTRY;
  type PORT_ENTRY is
    record
      ALT_ROUTING : BOOLEAN := FALSE;
      PORT : LOGICAL_LINE;
      ALT_PORT : LOGICAL_LINE;
      NEXT : ENTRY_PTR;
    end record;
  type PORT_LIST is array(INTEGER range <>) of LOGICAL_LINE;
  subtype RI_STRING is STRING(1..7);
  type RI_LIST_ELEM;
  type RI_PTR is access RI_LIST_ELEM;
  type RI_LIST_ELEM is
    record
      RI:RI_STRING;
      NEXT_RI:RI_PTR;
    end record;
  subtype RELAY_STRING is STRING(1..4);

  procedure READ_RI_TABLE(RI:RI_STRING; PORTS:out PORT_LIST;
      SUCCESS:out BOOLEAN);
  -- READ THE RI TABLE AND RETURN THE LOGICAL LINE(S) FOR A
      -- SPECIFIC RI
  -- SUCCESS WILL BE SET FALSE IF THE RI IS NOT FOUND

  type CHECK is (NOT_FOUND,FOUND);
  function CHECK_RI(RI:RI_STRING)return CHECK;
  -- CHECK FOR THE PRESENCE OF AN RI IN THE RI TABLE

  procedure ADD_RI(RI:RI_STRING;PORTS:ENTRY_PTR);
  -- ADDS AN RI TO THE APPROPIATE TABLE
  -- THE PORT ENTRY LIST FOR ANY RI OTHER THAN A COLLECTIVE RI
      -- WILL ONLY HAVE ONE ENTRY

  procedure DELETE_RI(RI:RI_STRING);
  -- REMOVES AN RI FROM THE TABLES
```

```
   procedure CHANGE_RI(RI:RI_STRING;LINE:LOGICAL_LINE);
   -- SETS THE LOGICAL LINE FOR A NON_COLLECTIVE RI

   procedure CHANGE_COLLECTIVE_RI(RI:RI_STRING;OLD_LINE,
       NEW_LINE:LOGICAL_LINE);
   -- CHANGES THE PORT ENTRY FOR OLD_LINE TO NEW_LINE
   -- IF OLD_LINE = 0, ADDS A PORT_ENTRY
   -- IF NEW_LINE = 0, DELETES THE PORT_ENTRY FOR THE OLD_LINE

   procedure CHANGE_ALT(RI:RI_STRING; ALT_LINE:LOGICAL_LINE;
       START:BOOLEAN:=FALSE);
   -- CHANGES THE ALT ROUTING FOR A NON_COLLECTIVE RI
   -- IF START = TRUE, BEGINS ALT ROUTING

   procedure CHANGE_COLLECTIVE_ALT(RI:RI_STRING;LINE,
       ALT_LINE:LOGICAL_LINE; START:BOOLEAN:=FALSE);
   -- CHANGES THE ALT ROUTING FOR ONE PORT ENTRY OF A COLLECTIVE
       -- RI
   -- IF START = TRUE, BEGINS ALT ROUTING FOR THIS PORT ENTRY

   procedure BEGIN_ALT(RI:RI_STRING);
   -- STARTS ALT ROUTING FOR A NON-COLLECTIVE RI

   procedure BEGIN_COLLECTIVE_ALT(RI:RI_STRING;LINE:LOGICAL_LINE);
   -- STARTS ALT ROUTING FOR ONE PORT ENTRY OF A COLLECTIVE RI

   procedure END_ALT(RI:RI_STRING);
   -- ENDS ALT ROUTING FOR A NON-COLLECTIVE RI

   procedure END_COLLECTIVE_ALT(RI:RI_STRING;LINE:LOGICAL_LINE);
   -- ENDS ALT ROUTING FOR A PORT ENTRY OF A COLLECTIVE RI

   procedure LOAD_RI_TABLE;
   -- LOADS THE RI TABLES FOR START UP

   procedure SAVE_RI_TABLE;
   -- SAVES THE RI_TABLE FOR RESTARTS

   procedure INIT_RI_TABLE;
   -- SETS UP AN EMPTY RI_TABLE
end RI_OPS;
```

## 3.5 Rationale for Hardware/Software Partitioning

Partitioning of the system functions into hardware and software was done with the primary goal of maximizing the system efficiency and a secondary goal of minimizing the amount of hardware. Particular implementations were not selected but rather each function was examined for suitability of implementation in hardware or software.

In general, most of the functions, or operations on objects, were partitioned into software while the objects themselves or object representations were partitioned into hardware. For example, translation of messages was selected for software implementation while the messages themselves are represented by bit-patterns in some form of memory (hardware).

The exceptions to this rule are the actual transmit and receive functions for both synchronous and asynchronous formats. These functions include bit stream operations and serial/parallel conversions. The processes are well defined, time consuming, and mechanical and are well suited to hardware implementation. In addition, there are many integrated circuit devices available which implement these functions with a minimum of additional hardware and software.

Since the development of the Message Switch proceeded directly from the system specification and requirements document, it evolved into a functional system which operated on a group of "objects". As such, it was independent of any preconceived hardware system. This allowed the designers to consider various hardware approaches for implementation. Three basic hardware configurations were examined: 1) a single large central processing machine, 2) a distributed (several smaller machines) and 3) a distributed hierarchical (single medium sized machine with several smaller distributed machines).

The centralized configuration was discarded for two reasons. First, the designers felt that it was unrealistic to expect Ada to be able to handle the large number of tasks which would be required (possible one thousand) in one machine. And, second, it is doubtful that a processor is available with the necessary speed and power to handle a 50 line switch, with the possible exception of a Cray.

The distributed configuration was considered in two variations, a vertical and a horizontal. For example: putting Input in one processor, Processing in another, and Output in a third, as opposed to say, ten inputs, outputs, and message processing in each of five processor units. While there are some attractive prospects to both of these variations, the disadvantages outweighed the potential benefits. The disadvantages included excessive interprocessor communication, forced sharing of tables and

status information across processor boundaries, and the possibility of multiple 16K bit per second channels overloading a single processor.

The designers agreed that the distributed hierarchical system is the best configuration for a message switch application. The logical result of the hardware/software partitioning, as related to message output is illustrated in Figure 3.5-1. The design consists of a combination of the two variations of the distributed approach, and as such, allows the designer to take advantage of the strengths of both. It was decided that Input and Output processing would be maintained on a lower level compared to the other more generalized and less time critical processes. At the same time Input/Output is to be spread horizontally over several processors to minimize the chances of overloading any single processor. Combining Input and Output in the same processor shortens the lines of communication between the two processes and improves the channel Transmit and Receive coordination. An additional advantage is availablity, a single point failure would disable no more than eight channels.

Of course there are some disadvantages but they can be minimized. The system will require more processors and interprocessor communication will be needed, causing an increased number of interconnections.

Because of the reduced real time constraints of a distributed system, the first disadvantage is minimized because smaller and less expensive processors can be utilized. There is certainly increased overhead because of interprocessor communication, but this can be minimized by judicious partitioning of the functions. The other disadvantage of interprocessor communications, which is the additional hardware interconnection, can be minimized by using a medium speed serial data bus instead of a parallel bus.

Because it was not known how much support would be provided by the Ada run time package, the configuration shown in Figures 3.5-2 and 3.5-3 were developed. This method would not depend on Ada run time support, but unfortunately, becomes quite machine dependent and reduces transportability. Additional logic was added to handle serial communication protocol and to provide low level support functions that were no longer available in common memory.

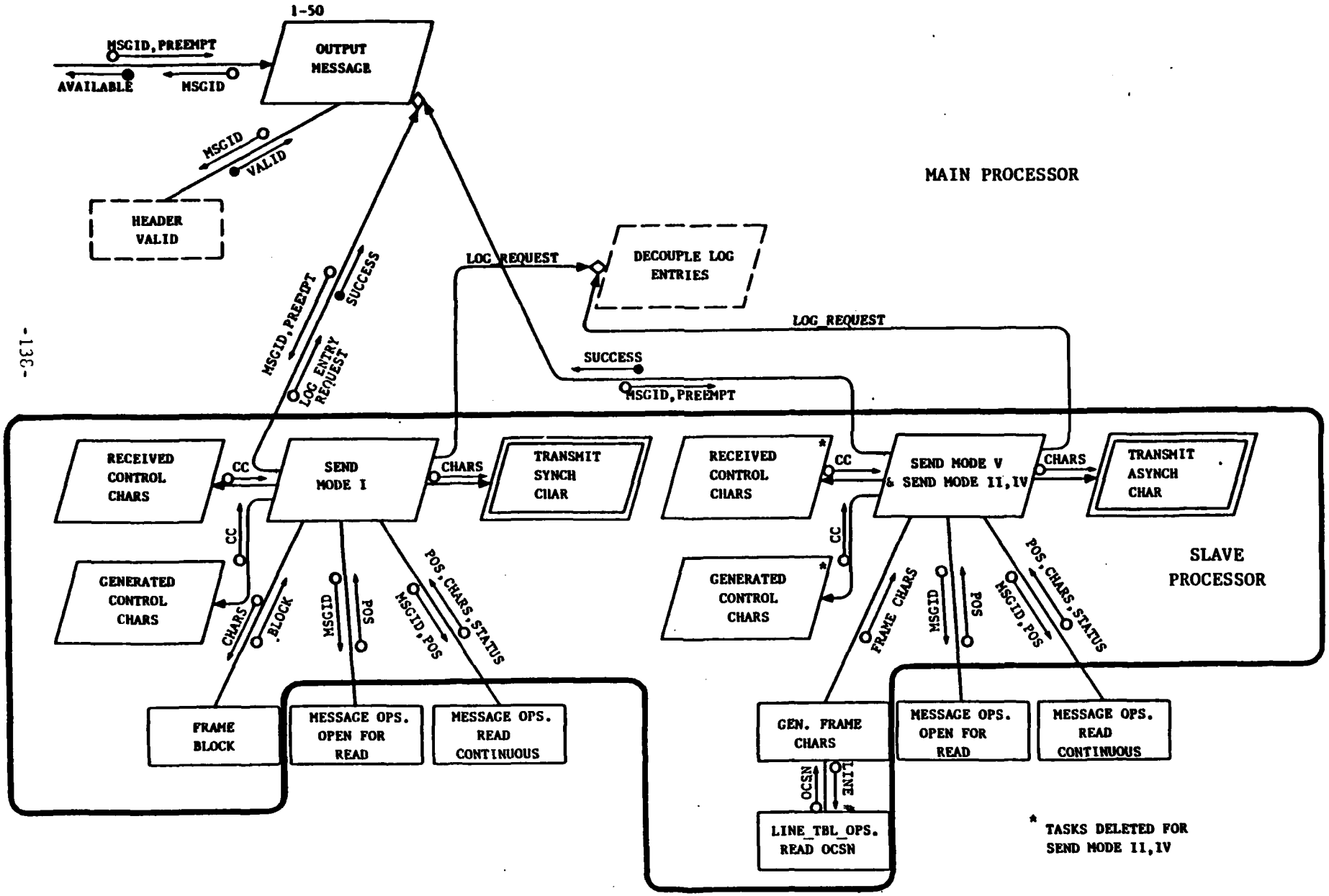



Figure 3.5-1 Logical Result of Hardware/Software Partitioning

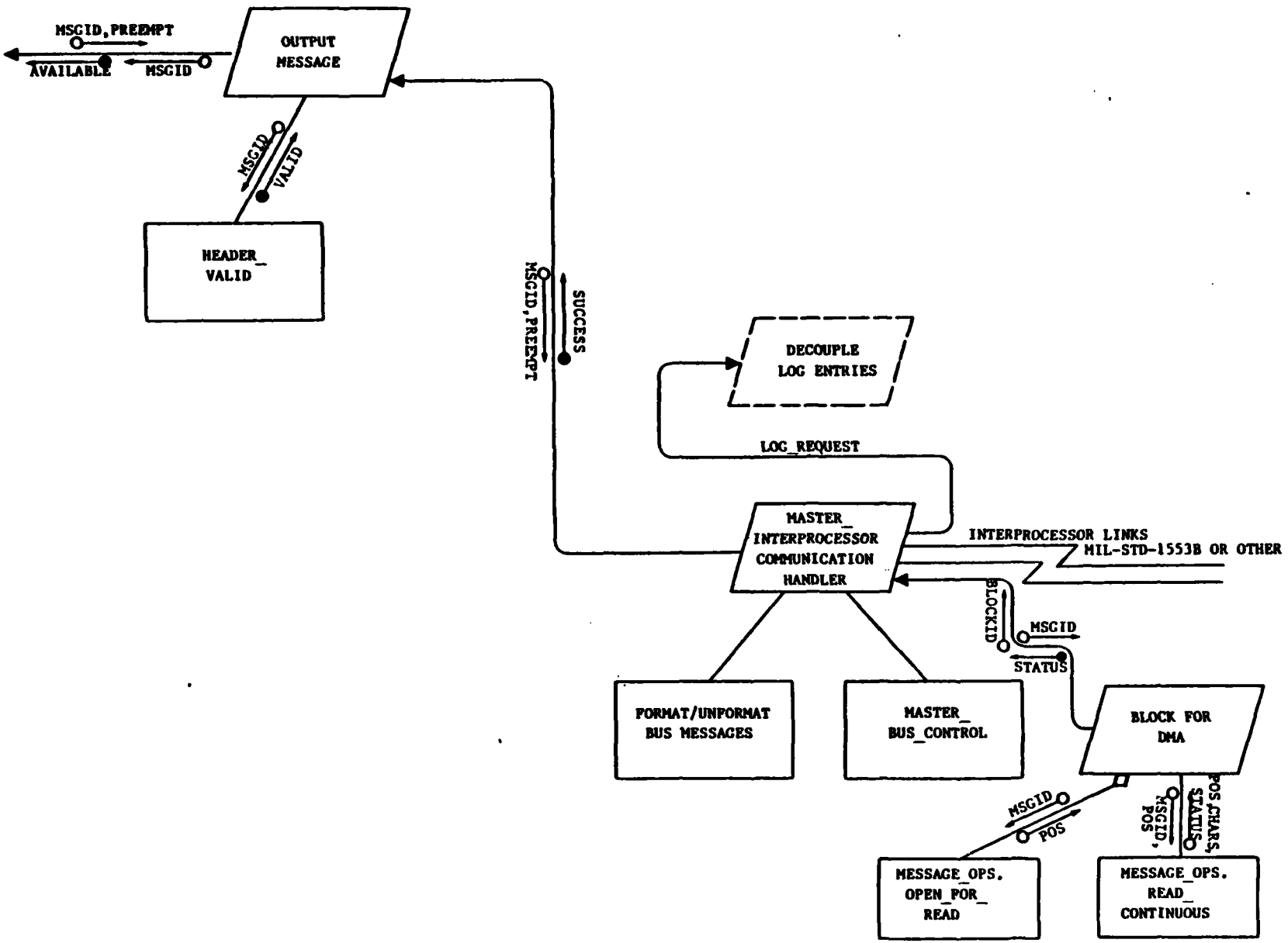
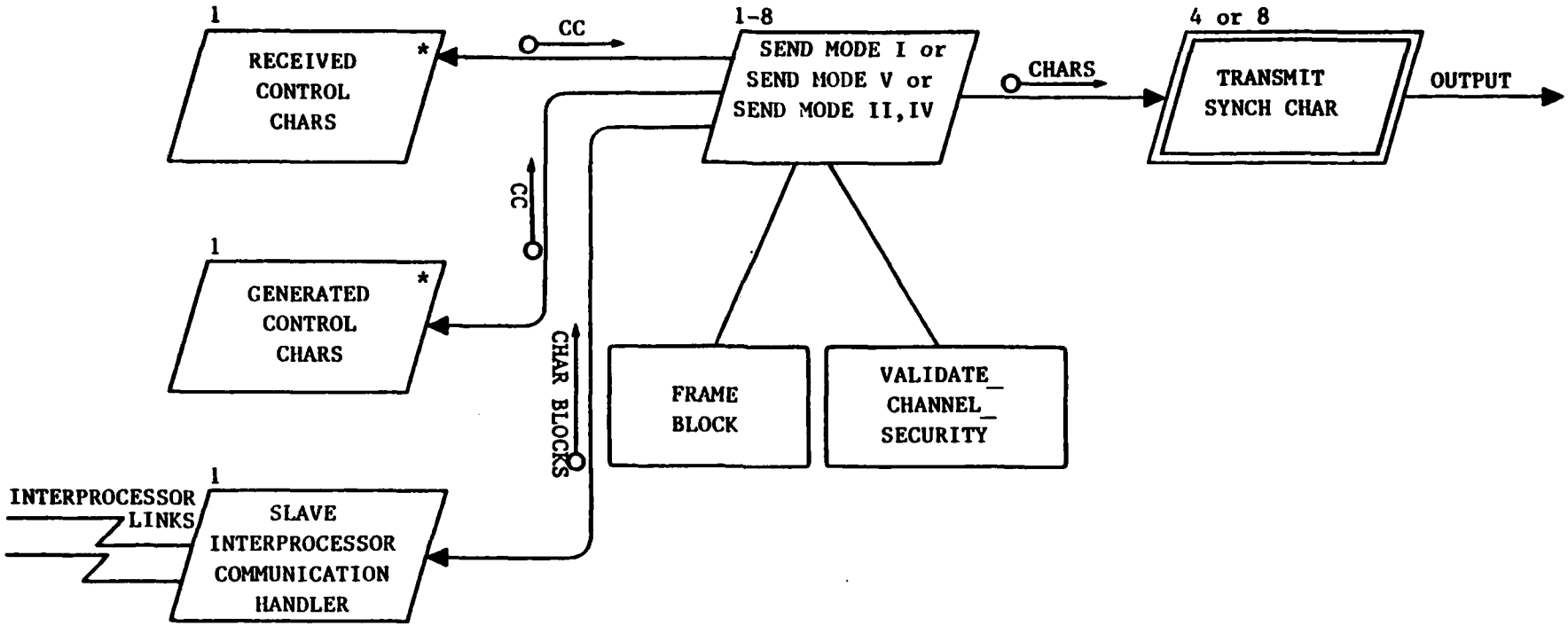



Figure 3.5-2 "OUTPUT MESSAGE" MAIN PROCESSOR SOFTWARE DIAGRAM




* TASKS DELETED FOR SEND MODE II,IV

Figure 3.5-3 "OUTPUT MESSAGE" REMOTE PROCESSOR SOFTWARE DIAGRAM

# 4. Detailed Hardware Design

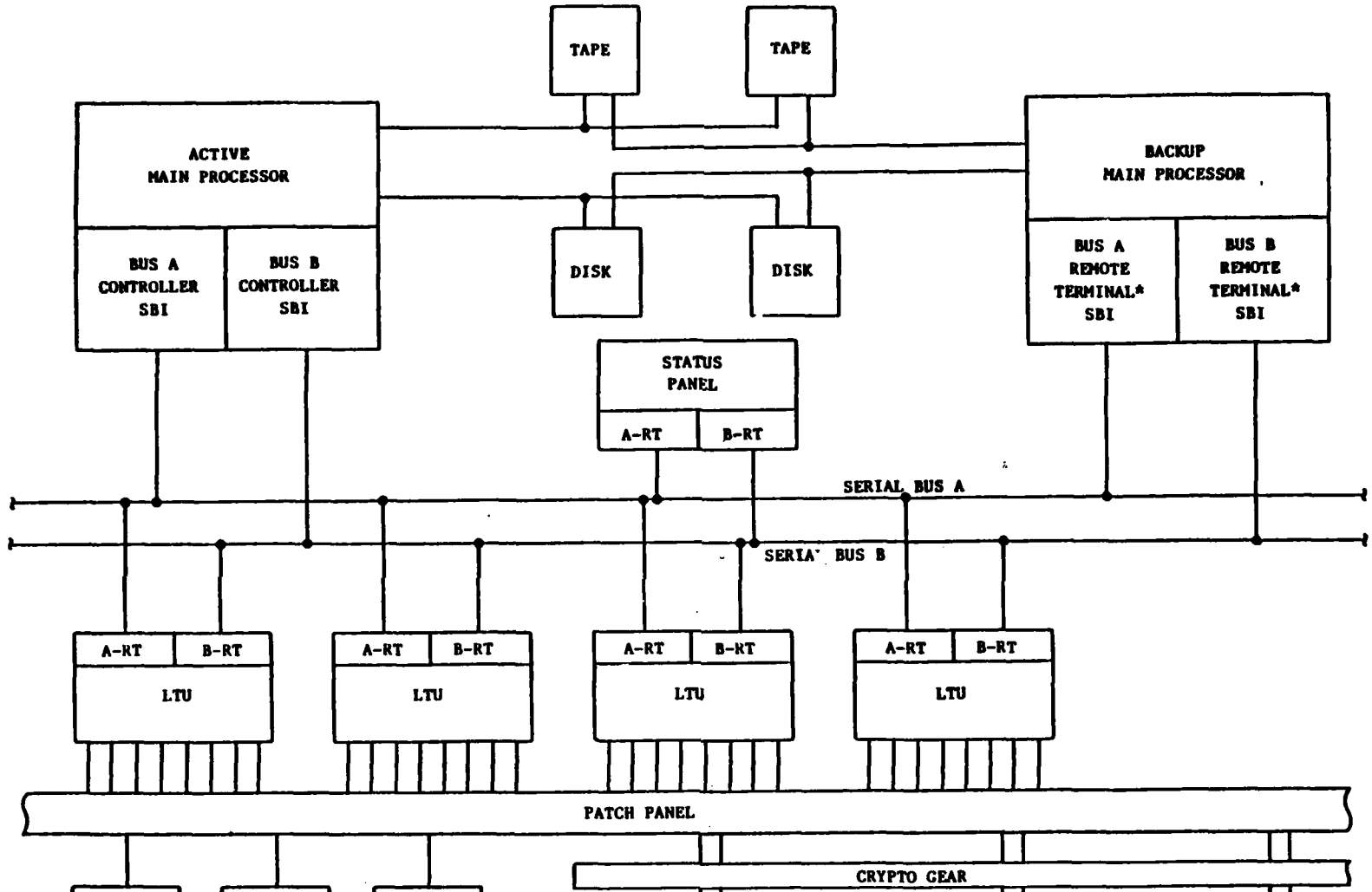
## 4.1 General Configuration

Figure 4.1-1 illustrates the switch system design, consisting of a generalized computing system that is optimized for real-time communications processing. In normal operation incoming messages are received by the Line Termination Units (LTUs), which handle the line level protocol functions. The messages are assembled into segments for transmission to the active main processor. At the main processor the segments are logged, saved on reference storage, translated and routed. After routing, the messages are transferred to the appropriate LTU(s) for final validation before transmission to the appropriate stations.

Start-up of the system takes place by operator action and includes loading of the operational programs from nonvolatile media. Database tables may be loaded from nonvolatile media by operator entry. Operational programs for the LTUs are down-loaded via the serial data bus. All of the processors contain a "bootstrap" and diagnostic program in Read Only Memory.

The main processor may be any general purpose machine with the necessary speed and I/O capability. Some special adaptation will be required to facilitate interprocessor monitoring and switchover. Magnetic tape drives are included as mass storage peripherals for sequential storage applications such as journals and intercepted messages. The disk drives are intended to be used as main program storage for start-up, mass random access storage for intransit messages, reference storage and journals. The specification requires that intransit storage be of sufficient size to contain 2500 average length messages for a 50 line switch. This is 6,000,000 characters, excluding labels and linkage. Although a memory of this size is possible in direct access Read/Write Memory (such as semiconductor RAM) it is not as cost effective as using large capacity disks. Both disk and tape peripherals are backed-up with a redundant unit to prevent single failures from causing a system outage.

The purpose of the line termination units (LTU) are to provide time critical protocol analysis and oversee the data send/receive function for each of the possible fifty channels of the message switch. Each LTU contains a general purpose microcomputer which is capable of being downloaded with channel dependent software. There also exists a read-only memory (ROM) containing the bootstrap program for start-up of the serial bus and program down-loading, as well as diagnostics and fault detection firmware. The LTUs contain the necessary circuitry for electrical interfacing, controlling, and monitoring crypto and modems associated with a channel.




* REMOTE TERMINAL SBI IN BACKUP PROCESSOR WILL BE DYNAMICALLY RECONFIGURED TO A CONTROLLER WHEN THAT PROCESSOR IS ACTIVATED.

Figure 4.1-1 MESSAGE SWITCH HARDWARE BLOCK DIAGRAM

## 4.2 Serial Data Base

A redundant data bus is required to prevent single failures from rendering the system inoperative. Physical constraints associated with a redundant bus virtually eliminate a parallel bus from consideration. Primarily, the large number of connections required to propogate a parallel bus from circuit board to circuit board, and the associated cabling problems are prohibitive. A serial bus eliminates these problems by requiring that only two cables be connected to each device for data bus access.

The ideal solution in the interconnect area as well as the areas of throughput and error is to use the MIL-STD-1553B serial bus. This selection has the added benefit of reducing risk because of its established technology. The serial bus is used to transfer message segments, control, request, and status information between the Main processor and the LTUs, as well as updating the Status Panel Displays and down-loading the operational programs to the LTUs. The 1553B Serial Data Bus operates at a 1 MHz bit rate and is capable of transferring 480 bytes of data plus a 32 byte label in 5.760 microseconds, including overhead and polling time. This is a bus capacity of 81,081 characters (bytes of data) per second. The specification requires a 50 line switch to be capable of handling a maximum of 9,000 characters in any one second period. Therefore, as long as bus overhead stays below 8 times the maximum data handling requirement the serial data bus will be equal to the task. Standard serial bus modules are available commercially and are designed to use DMA capabilities to reduce the required processor interactions.

The interprocessor message routing is determined by the physical port number. LTU1 contains ports in the range 1-8, LTU2 range 9-16, etc. Although it is not necessary that all consecutive ports be implemented or defined, maximum use can be made of the hardware by defining the ports in groups of four. This is because four channels are supported by each interface printed circuit board (PCB).

## 4.3 Line Termination Unit

Each LTU shown in Figure 4.3-1 is a two or three board set consisting of a CPU Board and one or two Channel Boards. Each channel board terminates four lines (channels), for a maximum of eight per LTU. This flexibility is provided so that increased processing power is available for high speed or complex channel protocol. On the other hand, a larger quantity of low speed channels in a grouping can also be accommodated by adding the extra channel board. The main determining factors are the processor's speed and efficiency in handling the channels.

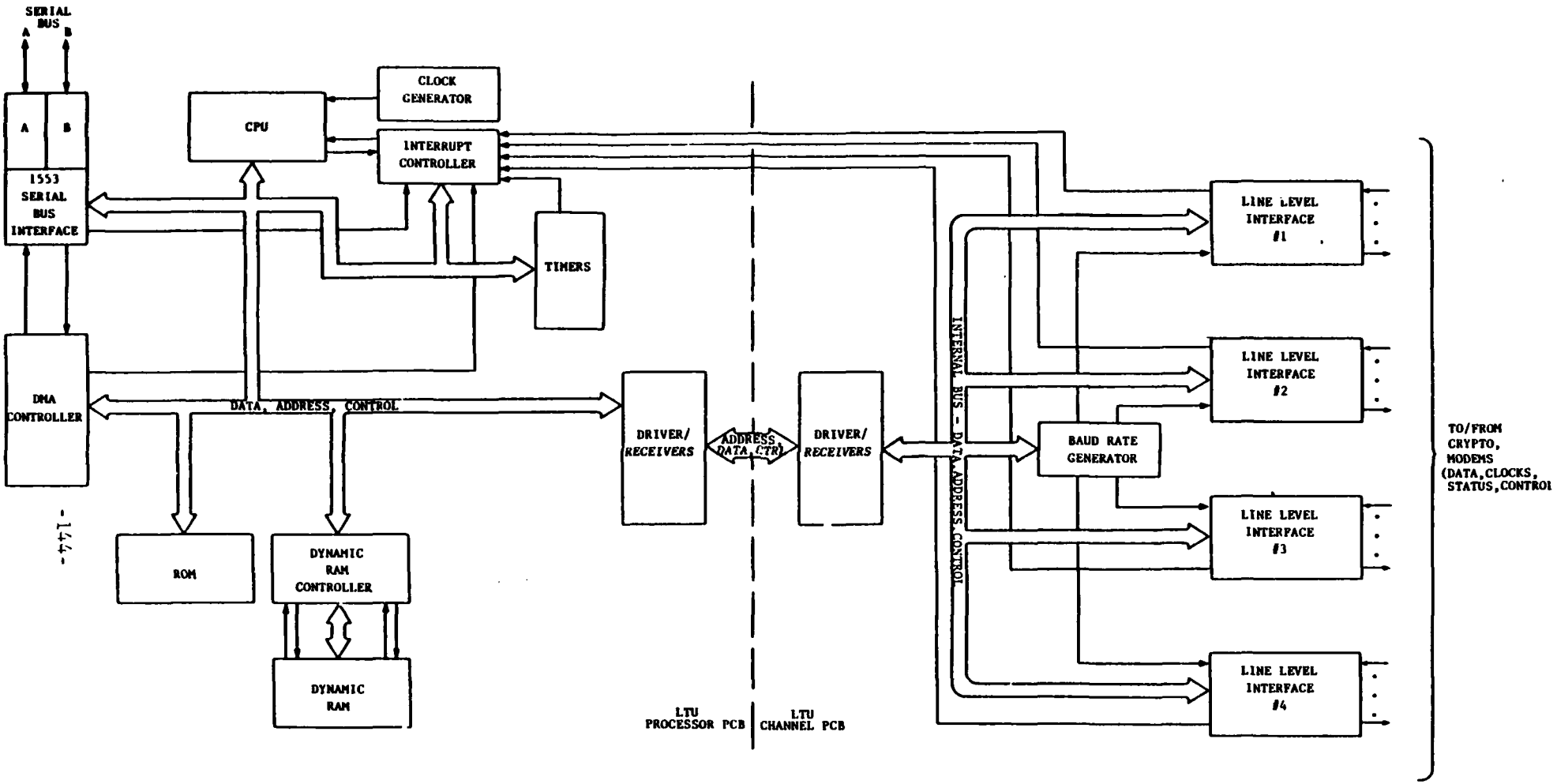



Figure 4.3-1 LTU BLOCK DIAGRAM

Each LTU is equipped with its own dual serial bus interface and DMA Controller for transferring data and programs directly to and from an on board dynamic random access memory (RAM). Each channel interface, which consists of serial data, control, clocks, and status lines, has an associated baud rate generator and interrupt lines. Other functional blocks are self explanatory and standard for a microprocessor system.

## 4.4 Design Features

Several features of this design are worthy of mention. Although it is expected that one main processor is sufficient for operational needs, a second processor is included for backup and monitoring purposes. The serial bus terminals of this processor are initially configured as a remote terminal but are capable of dynamic reconfiguration to the bus controller mode when processor switch-over occurs. Also at the time of processor switch-over the failed processor is forced to relinquish control of the mass storage peripherals to the backup processor.

The dual serial bus is redundant with automatic error detection capability. Each of the two serial busses is capable of handling the full specified message load plus overhead.

The final important feature is the treatment of operator terminals and local equipment. To prevent the proliferation of special purpose interfaces this equipment is interfaced through Line Termination Units in the same manner as normal serial communication channels.

## 5. Implementation of Selected Function (Output Message)

### 5.1 Hardware Implementation

In the system design process outlined in the Ada Integrated Methodology the hardware design and implementation are a part of the detailed design phase of system development. As the Ada Capability Study progressed, the output component was selected as the module to be designed in detail and programmed in Ada. For this reason, more attention was given to the design of the LTUs than to the main processor. In the design process, hardware was selected which has performed well in similar communications applications. It is not possible at this time to know for sure whether Ada compilers and an Ada language system will support the hardware selected.

For processing power and flexibility, an 8-bit CPU such as an INTEL 8088 is recommended. This would allow for upgrade to a 16-bit processor such as the 8086, if additional processing power is needed. Such an upgrade would increase the requirements for RAM, ROM, and driver ICs, but would simplify the serial bus interface controller because the bus operates on 16-bit words.

The following sections describe the two printed circuit boards (PCB) of the LTU in more detail.

### 5.2 LTU Processor PCB

This PCB, illustrated in Figure 5.2-1, comprises the elements that are common to all of the controlled serial channels, such as the serial bus interface, CPU, RAM, and ROM. The only item which is unusual in the context of a general purpose microcomputer PCB is the ID Register. This register is a set of "three-state" bus drivers which are readable by the CPU. The inputs to the drivers will be connected, through the board connector, to either +5 volts or ground. Each Processor PCB board slot will be programmed (hand-wired on the back-plane) with a unique combination of logic so that each processor can determine its own address. This address will be the one that the Serial Bus Controller will use when communicating with each processor. The Processor PCB also contains the bus driver and receivers necessary to communicate with one or two channel PCBs.

Preliminary estimates of parts count and area indicate that the ICs required will fit on a standard 77 square inch PCB, with sufficient margin to allow for the processor upgrade mentioned earlier.

### 5.3 LTU Channel PCB

This PCB, as illustrated in Figure 5.3-1, contains the circuitry necessary to interface to the serial channels and

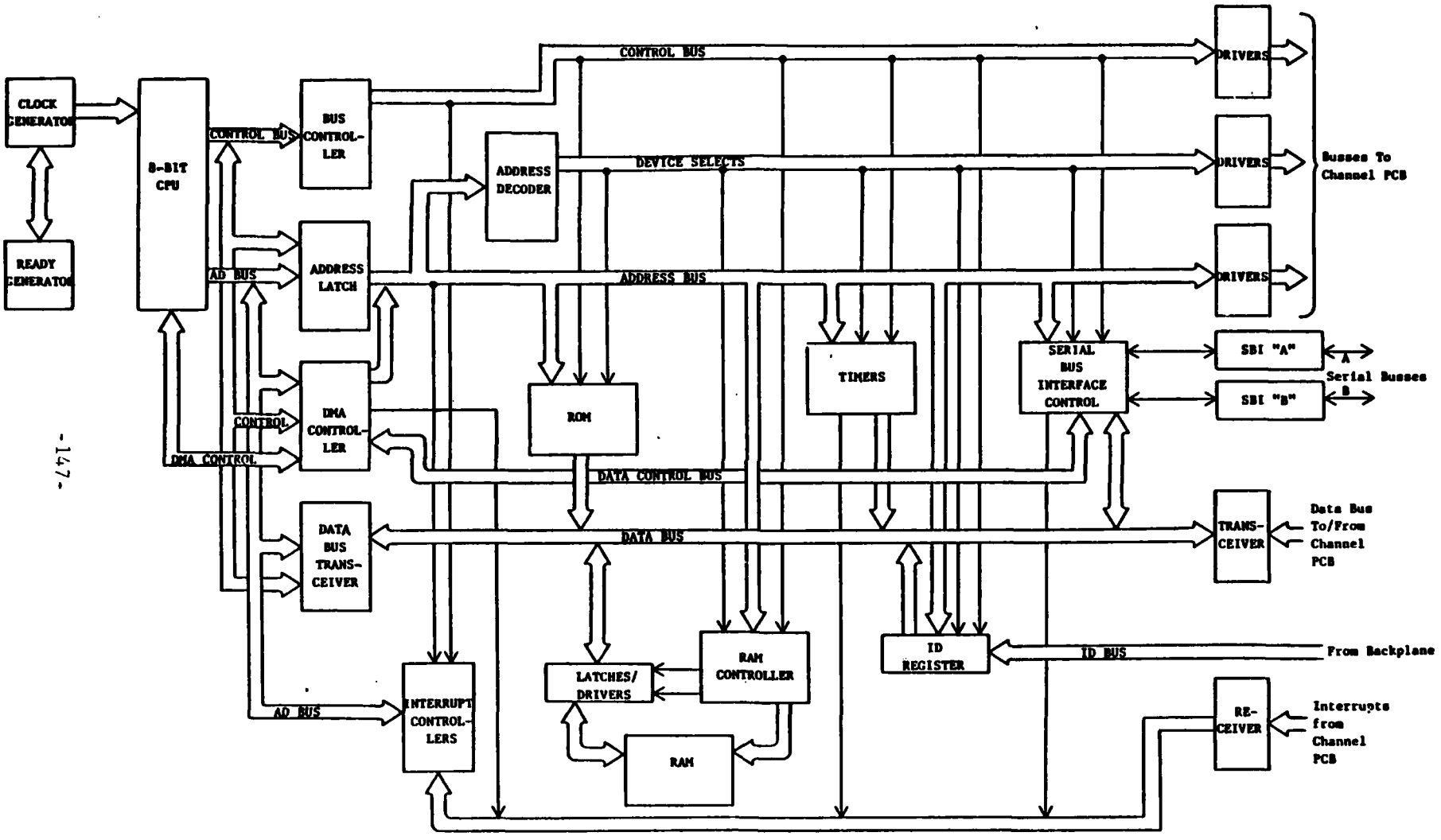



Figure 5.2-1 LTU PROCESSOR PCB BLOCK DIAGRAM

the associated channel equipment (crypto and/or modems), as well as baud rate generator, and the processor bus interface circuits.

Circuitry for four serial channels is provided on each PCB. The serial channel is terminated with a programmable USART which has the capability to transmit and receive both synchronous and asynchronous data. A Register is provided to latch control outputs for crypto and modems, as well as a register to receive status. All of these are provided with electrical interfaces to perform level shifting and isolate the external lines from the logic circuits.

The Baud Rate Generator is programmable and can provide different clock rates for any of up to four asynchronous channels.

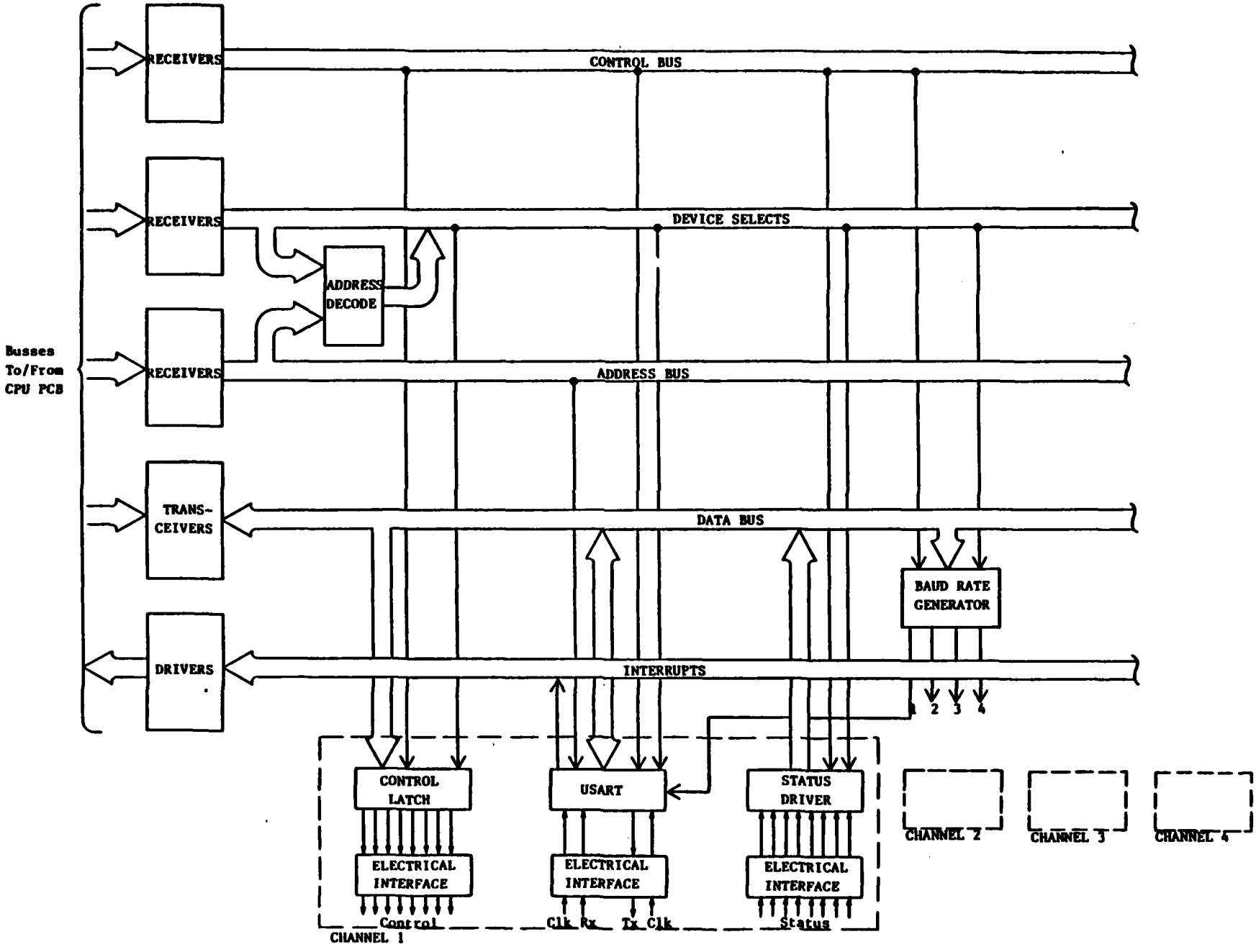



Figure 5.3-1 LTU CHANNEL PCB BLOCK DIAGRAM

## 5.4 Software Implementation

The software implementation for this project was not complete due to the allowed time and budget. Since the message output section was coded in Ada, the intended switch initialization and operation is oriented toward that portion of the switch.

### 5.4.1 Switch Operation

The following sequence, although not implemented, is intended to show how the message output tasks are created and become ready to output actual messages through the hardware. A key feature of this deisgn is the dynamic allocation and deallocation of output tasks in order to support database changes input by the operator while the switch is in operation.

Power Up Link protocol and bootstrap are stored in ROM in both processors, so the bus is capable of running on power up. However, no "output message" or "send sync/async" tasks are yet defined until the database is loaded.

Switch Initialization Part of the job of the switch initialization routine is to run a short diagnostic to determine the number of operational remote links. This process would thus determine the number of physical ports available in the system and print a copy on the printer. The configuration is determined by hard wiring of the connectors into which the link protocol and port printed circuit boards are inserted.

Data Base for Output Message The site dependent parameters (database) are entered interactively from the keyboard of a video display unit (VDU) or from a pre-prepared magnetic media such as tape or disk under the control of the "run switch" module.

The VDU command "define physical port" will prompt the user for the port number (or range of numbers) to be defined. In addition, information such as channel mode, transmission rate (baud), security level, etc. will be entered as requested. At the conclusion of each port definition sequence, the prompt: "Is this information correct (Y or N)?" will appear. If not correct the sequence will be repeated until the information is correct, at which time the following sequence will occur:

1. Run switch will lock out users of the "line table", update the necessary line table information, and unlock the table in the main processor. Line table information needed by the remote processor will be transmitted over the data link to the remote processor corresponding to the physical

port number for entry in a local table.

2. Run switch uses the TASKS package to create a new output message task and initialize the task by issuing the physical port number. The TASKS package consists of an array of records of access types (task entry point for various output message routines) indexed by physical port number.

3. Run switch also uses the TASKS package to create and initialize the Send task (Mode I, II and IV, and V) and generate and receive control character tasks, if required, (depends on channel mode). This process is communicated over the interprocessor data link to the remote processor corresponding to the physical port being changed.

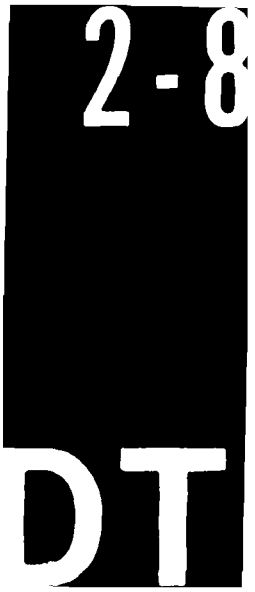
After the port is placed into active service, the channel is then ready to output message data.

An example of the calls from the "Run Switch" to the "Tasks" package in order to initialize physical line 1 for mode V operation is as follows:

```
TABLE(1).OUTPUT:= new OUTPUT_MESSAGE;   --create the output
                                        --message task.
TABLE(1).OUTPUT.INIT(1);                --tell output message
                                        --what physical line he is.
TABLE(1).SEND_V:= new SEND_MODE_V;      --generate mode V send task.
TABLE(1).SEND_V.INIT(1,UART_ADDRS);     --tell send task what UART
                                        --to use.
TABLE(1).GENCC:= new CONTROL_CHARACTER;--create the cntl char gen tsk
TABLE(1).GENCC.INIT(1); --tell cntl char tsk where to receive chars
TABLE(1).RECVCC:= new CONTROL_CHARACTER;
TABLE(1).RECVCC.INIT(1); -- what line number is this
```

### 5.4.2 Ada "Output Message" Code

This code is contained in a separate document due to its length.


2-8
DT